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
Power Family
RISC/System 6000®
POWER®
POWER2
POWER4
POWER4+
POWER5
POWER5+
POWER6®
POWER7®
POWER8®
POWER9™
System/370
System z

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

The roots of the Power ISA (Instruction Set Architecture) extend back over a quarter of a century, 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 IIIIs) and optional application-specific facilities (categories) as extensions to a pervasive base architecture.

Power ISA Version 3.0 B focuses 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 have been eliminated to ensure increased application portability between Power processors. Legacy embedded applications that require the eliminated material will continue to use V. 2.07B.

The Power ISA Version 3.0 B 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 2.07B. Change bars may be omitted for changes associated with removing obsolete categories and the second Book III.
Summary of Changes in Power ISA Version 3.0 B

This document is Version 3.0 B of the Power ISA. It is intended to supersede and replace version 2.07B. 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 2.07B. Change bars in this summary of changes indicate new, changed, or removed changes relative to V. 3.0.

Instruction Fusion: Specifies instruction sequences that, when placed consecutively in the program, are expected to provide improved performance.


Decimal Integer Support Operations: Adds new BCD support instructions, including variable-length load/store instructions for bcd values, new format conversion instructions between BCD and National decimal, zoned decimal, and 128-bit signed integer formats, new BCD truncate, round, and shift instructions, new BCD sign digit manipulation instructions. Also adds multiply-by-10 instructions to facilitate binary-to-decimal conversion for printf. Corrected functionality of Decimal Shift and Round (bcdsr.) instruction.

Decimal Floating-Point Support Operations: Add immediate forms of DFP Test Significance instructions.

Binary Floating-Point Support Operations: Adds new binary floating-point support instructions (e.g., exponent and significand extraction and insertion) to enhance implementation of math libraries.

Quad-Precision Binary Floating-Point Operations: Add new instructions to support IEEE-754-2008 binary128 floating-point.

String Operations (FXU option): Adds instructions to accelerate character testing functions.

String Operations (VSU option): Adds instructions to accelerate string processing and targeted character extraction.

Vector Half-Precision Floating-Point Support Operations: Adds support for IEEE-754-2008 binary16 floating-point as a transport format.


128-bit SIMD FXU Operations: Adds remaining 32-bit and 64-bit FXU functionality to vector instruction set.

128-bit SIMD Miscellaneous Operations: Enhances support for Little-Endian processing with new load/store instructions and new permute-class instructions, new byte and halfword element load/store instructions, and vector element insertion/extraction.

System Call Extension: Provides a new form of system call that can direct execution to one of a number of locations and that provides other enhancements.

PC-Relative Addressing: Specifies a new instruction that adds an immediate value to the program counter and writes it to the destination register in preparation for use with a D-Form Load instruction.

Hypervisor msgsnd Instruction Enhancements: Extends the msgsnd instruction so that messages can be sent throughout the system.

Performance Monitor Enhancements: Reserves a special no-op instruction for use by the Performance Monitor, and increases the scope of control of the Performance Monitor bit of the Hypervisor Facility Status and Control register.

Radix Tree and Related MMU Extensions: Adds support for the radix tree style of MMU with full virtualization and related control mechanisms that manage its coexistence with the HPT. Also adds a tblie variant that invalidates multiple consecutive translations.

Copy-Paste Facility: Adds support for a new facility that enables an application to initiate accelerator operations.

Optimizing mtspr Sequences: Reserves an SPR to be used in a no-op mtspr to indicate the beginning of a sequence of mtsprs that can be done without synchronizing each one independently.

Atomic Memory Operations: Adds support for a new facility that performs simple atomic operations directly in memory to avoid bringing the line through the cache hierarchy when another core is likely to be the next user.

Event-Based Branch Extension: Adds External Event-Based Branch exception and status bits to the BESCR.

Processor Compatibility Register: Adds a new V 2.07 bit to the PCR that controls the availability facilities in problem state that are introduced in this level of the architecture.

Atomicity and Alignment Enhancements: Limits the number of disjoint atomic storage accesses that are allowed for various non-atomic storage accesses.

Power-Saving Mode: Replaces the existing power-saving mode instructions with a single stop instruction, and enables the operating system to enter a limited set of power-saving levels without hypervisor involvement.

D-form VSX Floating-Point Storage Access Instructions: Adds base+displacement forms of VSR load and store instructions.
Integer Multiply-Add Instructions: Adds new integer multiply-add instructions to accelerate arbitrary-length multiplication.

**msgsndp** Hypervisor Facility Availability Interrupt: Adds a new HFSCR bit to control the availability of the **msgsndp** instruction and the associated control registers.

VSX Permute: Adds new pernute instructions that can address all 64 VSRs.

Array Index Support: Enhance support for mixed-data-type addressing into arrays (e.g., base + 32-bit index)

Hypervisor Virtualization Interrupt: Defines a new exception and corresponding interrupt that is caused by events external to the processor that relate to virtualization.

**wait** Instruction Enhancements: Improves the capabilities of the **wait** instruction so that resumption of processing can occur due to event-based branches and external signals.

Decrementer and Hypervisor Decrementer Enhancements: Defines a new mode bit in the LPCR that enables additional Decrementer and Hypervisor Decrementer bits in order to increase the time between the associated interrupts.

Deliver A Random Number: Adds a new instruction to place a random number in a GPR in one of three formats.

Data Storage Interrupt Status Register for Alignment Interrupt: Simplifies the Alignment interrupt by removing the Data Storage Interrupt Status Register (DSISR) from the set of registers modified by the Alignment interrupt.

CA32 & OV32 and Move XER to CR Extended: Added support for 32-bit CA & OV status in 64-bit mode for dynamically-typed languages.

VSX Shift Variable: Accelerate parallel element extraction from packed vectors of arbitrary-width-element values.

Enhanced Virtualization for Linux: Delivers exceptions caused by the OS attempting to use hypervisor instructions and SPRs to the hypervisor instead of the OS.

Accesses to unimplemented SPRs by the OS newly cause interrupts that are also directed to the hypervisor.

Synchronizing Messages and Storage Updates: Adds a new instruction to make latent storage updates from another thread accessible after receiving a Directed Hypervisor Doorbell interrupt from that thread.

VSX Conditional: Adds new instruction to accelerate conditional, maximum, and minimum operations. Withdrew xscmpnedp, xvcmpnsp[.], and xvcmpnedp[. ] instructions introduced in v3.0.

FXU & Vector Extensions for Blockchain Support: Two new instructions (addex and vmsumudm) introduced to accelerate arbitrary-precision integer arithmetic, and specifically to accelerate Blockchain's implementation of elliptical curve encryption signature algorithm. The OV bit is employed to provide an additional, independent carry status bit, allowing software to parallelize carry propagation.

Miscellaneous Changes: Makes minor clarifications, corrections, and editorial enhancements.

FX/VSX/Vector Miscellaneous: Editorial cleanup of Book I chapters 4, 5, and 7.

TM Multithread Overflow: Adds a bit to TEX-ASR to enable software to differentiate single thread footprint overflow from that aggravated by multiple threads competing for footprint.

Lightweight mffs: Modifications of mffs to accelerate saving/setting/restoring floating-point environments (e.g., rounding modes, exception trapping enables) common in math libraries that require overriding the environment.
Preface

Summary of Changes in Power ISA Version 3.0 B

Table of Contents

Book I:

Power ISA User Instruction Set Architecture

Chapter 1. Introduction

Chapter 2. Branch Facility

Chapter 3. Fixed-Point Facility

Table of Contents
6.9.1.6 Vector Integer Negate Instructions .................................................. 293
6.9.2 Vector Extend Sign Instructions ......................................................... 294
6.9.2.1 Vector Integer Average Instructions ................................................ 295
6.9.2.2 Vector Integer Absolute Difference Instructions ............................... 297
6.9.2.3 Vector Integer Maximum and Minimum Instructions ............................. 299
6.9.3 Vector Integer Compare Instructions ........................................................ 303
6.9.4 Vector Logical Instructions ................................................................. 312
6.9.5 Vector Parity Byte Instructions .............................................................. 314
6.9.6 Vector Integer Rotate and Shift Instructions .......................................... 315
6.10 Vector Floating-Point Instruction Set .......................................................... 321
6.10.1 Vector Floating-Point Arithmetic Instructions ........................................ 321
6.10.2 Vector Floating-Point Maximum and Minimum Instructions ..................... 323
6.10.3 Vector Floating-Point Rounding and Conversion Instructions .................... 324
6.10.4 Vector Floating-Point Compare Instructions .......................................... 328
6.10.5 Vector Floating-Point Estimate Instructions .......................................... 331
6.11 Vector Exclusive-OR-based Instructions .................................................. 333
6.11.1 Vector AES Instructions ....................................................................... 333
6.11.2 Vector SHA-256 and SHA-512 Sigma Instructions ................................. 335
6.11.3 Vector Binary Polynomial Multiplication Instructions .............................. 336
6.11.4 Vector Permute and Exclusive-OR Instruction ........................................... 338
6.12 Vector Gather Instruction .......................................................................... 339
6.13 Vector Count Leading Zeros Instructions ................................................... 340
6.14 Vector Count Trailing Zeros Instructions .................................................. 341
6.14.1 Vector Count Leading/Trailing Zero LSB Instructions .............................. 342
6.14.2 Vector Extract Element Instructions ..................................................... 343
6.15 Vector Population Count Instructions ....................................................... 345
6.16 Vector Bit Permute Instruction .................................................................. 346
6.17 Decimal Integer Instructions .................................................................... 347
6.17.1 Decimal Integer Arithmetic Instructions ............................................... 347
6.17.2 Decimal Integer Format Conversion Instructions ..................................... 350
6.17.3 Decimal Integer Sign Manipulation Instructions ..................................... 356
6.17.4 Decimal Integer Shift and Round Instructions ......................................... 357
6.17.5 Decimal Integer Truncate Instructions ................................................... 360
6.18 Vector Status and Control Register Instructions ........................................... 362

Chapter 7. Vector-Scalar Floating-Point Operations ............................................ 363
7.1 Introduction ................................................................................................. 363
7.1.1 Overview of the Vector-Scalar Extension ................................................ 363
7.1.1.1 Compatibility with Floating-Point and Decimal Floating-Point Operations 363
7.1.1.2 Compatibility with Vector Operations ................................................ 363
7.2 VSX Registers ............................................................................................. 364
7.2.1 Vector-Scalar Registers ............................................................................ 364
7.2.1.1 Floating-Point Registers .................................................................. 364
7.2.1.2 Vector Registers ............................................................................... 366
7.2.2 Floating-Point Status and Control Register ............................................. 367
7.3 VSX Operations ............................................................................................ 372
7.3.1 VSX Floating-Point Arithmetic Overview ................................................. 372
7.3.2 VSX Floating-Point Data .......................................................................... 373
7.3.2.1 Data Format ...................................................................................... 373
7.3.2.2 Value Representation ....................................................................... 375
7.3.2.3 Sign of Result .................................................................................. 376
7.3.2.4 Normalization and Denormalization .................................................. 377
7.3.2.5 Data Handling and Precision ............................................................. 377
7.3.2.6 Rounding ......................................................................................... 381
7.3.3 VSX Floating-Point Execution Models ..................................................... 384
7.3.3.1 VSX Execution Model for IEEE Operations ......................................... 384
7.3.3.2 VSX Execution Model for Multiply-Add Type Instructions ..................... 385
7.4 VSX Floating-Point Exceptions .................................................................... 387
7.4.1 Floating-Point Invalid Operation Exception ............................................ 390
7.4.1.1 Definition ......................................................................................... 390
7.4.1.2 Action for VE=1 ............................................................................... 390
7.4.1.3 Action for VE=0 ............................................................................... 392
7.4.2 Floating-Point Zero Divide Exception ...................................................... 401
7.4.2.1 Definition ......................................................................................... 401
7.4.2.2 Action for ZE=1 ............................................................................... 401
7.4.2.3 Action for ZE=0 ............................................................................... 402
7.4.3 Floating-Point Overflow Exception .......................................................... 404
7.4.3.1 Definition ......................................................................................... 404
7.4.3.2 Action for OE=1 ............................................................................... 404
7.4.3.3 Action for OE=0 ............................................................................... 407
7.4.4 Floating-Point Underflow Exception. 409
7.4.4.1 Definition ................. 409
7.4.4.2 Action for UE=1 .......... 409
7.4.4.3 Action for UE=0 .......... 411
7.4.5 Floating-Point Inexact Exception 414
7.4.5.1 Definition ................. 414
7.4.5.2 Action for XE=1 .......... 414
7.4.5.3 Action for XE=0 .......... 417
7.5 VSX Storage Access Operations . 420
7.5.1 Accessing Aligned Storage Oper-
    ands .................................. 420
7.5.2 Accessing Unaligned Storage Oper-
    ands .................................. 421
7.5.3 Storage Access Exceptions ... 422
7.6 VSX Instruction Set ............ 423
7.6.1 VSX Instruction Set Summary . 423
7.6.1.1 VSX Storage Access Instructions . 423
7.6.1.2 VSX Binary Floating-Point Sign Manipulation Instructions .............. 425
7.6.1.3 VSX Binary Floating-Point Arithmetic Instructions ................. 425
7.6.1.4 VSX Binary Floating-Point Compare Instructions .................... 428
7.6.1.5 VSX Binary Floating-Point Round to Shorter Precision Instructions .... 429
7.6.1.6 VSX Binary Floating-Point Convert to Shorter Precision Instructions .... 429
7.6.1.7 VSX Binary Floating-Point Convert to Longer Precision Instructions .... 429
7.6.1.8 VSX Binary Floating-Point Round to Integral Instructions ............ 430
7.6.1.9 VSX Binary Floating-Point Convert To Integer Instructions .......... 430
7.6.1.10 VSX Binary Floating-Point Convert From Integer Instructions ......... 431
7.6.1.11 VSX Binary Floating-Point Math Support Instructions ............... 431
7.6.1.12 VSX Vector Logical Instructions .......... 432
7.6.1.13 VSX Vector Permute-class Instructions ............................. 432
7.6.2 VSX Instruction Description Conven-
    tions ................................ 434
7.6.2.1 VSX Instruction RTL Operators 434
7.6.2.2 VSX Instruction RTL Function Calls ................................ 435
7.6.3 VSX Instruction Descriptions ... 480

Appendix A. Suggested Floating-Point Models ........ 775
A.1 Floating-Point Round to Single-Precision Model ............... 775
A.2 Floating-Point Convert to Integer Model ...................... 779
A.3 Floating-Point Convert from Integer Model .................... 782
A.4 Floating-Point Round to Integer Model ...................... 784

Appendix B. Densely Packed Decimal ......................... 787
B.1 BCD-to-DPD Translation .................. 787
B.2 DPD-to-BCD Translation .................. 787
B.3 Preferred DPD encoding ............... 788

Appendix C. Assembler Extended Mnemonics .................... 791
C.1 Symbols ................................ 791
C.2 Branch Mnemonics ...................... 792
C.2.1 BO and BI Fields ................. 792
C.2.2 Simple Branch Mnemonics .......... 792
C.2.3 Branch Mnemonics Incorporating
    Conditions ................................ 793
C.2.4 Branch Prediction ................... 794
C.3 Condition Register Logical Mnemonics ............... 795
C.4 Subtract Mnemonics .................... 795
C.4.1 Subtract Immediate .................. 795
C.4.2 Subtract ................................ 795
C.5 Compare Mnemonics .................... 796
C.5.1 Doubleword Comparisons .......... 796
C.5.2 Word Comparisons ................... 796
C.6 Trap Mnemonics ....................... 797
C.7 Integer Select Mnemonics ............. 798
C.8 Rotate and Shift Mnemonics .......... 799
C.8.1 Operations on Doublewords ........ 799
C.8.2 Operations on Words ............... 800
C.9 Move To/From Special Purpose Regis-
    ter Mnemonics ......................... 801
C.10 Miscellaneous Mnemonics ............. 802

Book II:

Power ISA Virtual Environment Architecture ........... 807

Chapter 1. Storage Model ....................... 809
1.1 Definitions ................................ 809
1.2 Introduction ........................... 810
1.3 Virtual Storage ......................... 810
1.4 Single-Copy Atomicity ............... 811
1.5 Cache Model .......................... 812
1.6 Storage Control Attributes .......... 812
1.6.1 Write Through Required .......... 813
1.6.2 Caching Inhibited .................. 813
1.6.3 Memory Coherence Required ...... 813
1.6.4 Guarded ............................ 813
1.6.5 Strong Access Order ............... 814
Chapter 1. Shared Storage
1.7 Shared Storage ........................................... 814
1.7.1 Storage Access Ordering .............................. 815
1.7.2 Storage Ordering of Copy/Paste-Initiated Data Transfers ........................................... 817
1.7.3 Storage Ordering of I/O Accesses .................. 817
1.7.4 Atomic Update ............................................ 817
1.7.4.1 Reservations ........................................... 818
1.7.4.2 Forward Progress ................................. 820
1.8 Transactions .................................................. 821
1.8.1 Rollback-Only Transactions ........................ 823
1.9 Instruction Storage ......................................... 823
1.9.1 Concurrent Modification and Execution of Instructions ........................................... 825

Chapter 2. Performance Considerations and Instruction Restart ........................................... 827
2.1 Performance-Optimized Instruction Sequences ........................................... 827
2.1.1 Load and Store Operations ......................... 828
2.1.2 32-Bit Constant Generation ....................... 831
2.1.3 Sign and Zero Extension ......................... 831
2.1.4 Load/Store Addressing Relative to Program Counter ........................................... 832
2.1.5 Destructive Operation Operand Preservation ........................................... 833
2.2 Instruction Restart ........................................... 834

Chapter 3. Management of Shared Resources ........................................... 835
3.1 Program Priority Registers ................................. 835
3.2 "or" Instruction ............................................. 835

Chapter 4. Storage Control Instructions ........................................... 837
4.1 Parameters Useful to Application Programs ........................................... 837
4.2 Data Stream Control Register (DSCR) ............... 837
4.3 Cache Management Instructions ...................... 839
4.3.1 Instruction Cache Instructions .................. 840
4.3.2 Data Cache Instructions .......................... 841
4.3.2.1 Obsolete Data Cache Instructions ........... 852
4.3.3 "or" Instruction .......................................... 853
4.4 Copy-Paste Facility .......................................... 854
4.5 Atomic Memory Operations ............................ 857
4.5.1 Load Atomic ............................................ 857
4.5.2 Store Atomic ............................................ 861
4.6 Synchronization Instructions ............................ 863
4.6.1 Instruction Synchronize Instruction ................ 863
4.6.2 Load and Reserve and Store Conditional Instructions ........................................... 866
4.6.2.1 64-Bit Load and Reserve and Store Conditional Instructions ........................................... 869
4.6.2.2 128-bit Load and Reserve Store Conditional Instructions ........................................... 871
4.6.3 Memory Barrier Instructions ...................... 873
4.6.4 Wait Instruction ........................................ 876

Chapter 5. Transactional Memory Facility ........................................... 877
5.1 Transactional Memory Facility Overview .......... 877
5.1.1 Definitions .............................................. 878
5.2 Transactional Memory Facility States .............. 880
5.2.1 Priority Levels .......................................... 882
5.2.2 Program启动s ....................................... 882
5.2.3 Transactional Memory State Register ............ 885
5.2.4 Transactional Memory State Register ............ 886
5.3 Transaction Failure .......................................... 882
5.3.1 Causes of Transaction Failure .................... 882
5.3.2 Recording of Transaction Failure ................ 885
5.3.3 Handling of Transaction Failure ................. 885
5.4 Transactional Memory Failure Registers ............ 886
5.4.1 Transaction Failure Handler Address Register (TFHAR) ........................................... 886
5.4.2 Transaction EXception And Status Register (TEXASR) ........................................... 886
5.4.3 Transaction Failure Instruction Address Register (TFIAR) ........................................... 889
5.5 Transactional Facility Instructions .................... 890

Chapter 6. Time Base ........................................... 897
6.1 Time Base Instructions ....................................... 898

Chapter 7. Event-Based Branch Facility ........................................... 901
7.1 Event-Based Branch Overview ........................... 901
7.2 Event-Based Branch Registers ....................... 902
7.2.1 Branch Event Status and Control Register ........... 902
7.2.2 Event-Based Branch Handler Register .......... 903
7.2.3 Event-Based Branch Return Register .......... 904
7.3 Event-Based Branch Instructions ....................... 905

Chapter 8. Branch History Rolling Buffer ........................................... 907
8.1 Branch History Rolling Buffer Entry Format .......... 908
8.2 Branch History Rolling Buffer Instructions .......... 909
# Appendix A. Assembler Extended Mnemonics

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.1 Data Cache Block Touch [for Store] Mnemonics</td>
<td>911</td>
</tr>
<tr>
<td>A.2 Data Cache Block Flush Mnemonics</td>
<td>911</td>
</tr>
<tr>
<td>A.3 Or Mnemonics</td>
<td>911</td>
</tr>
<tr>
<td>A.4 Load and Reserve Mnemonics</td>
<td>911</td>
</tr>
<tr>
<td>A.5 Synchronize Mnemonics</td>
<td>912</td>
</tr>
<tr>
<td>A.6 Wait Mnemonics</td>
<td>912</td>
</tr>
<tr>
<td>A.7 Transactional Memory Instruction Mnemonics</td>
<td>912</td>
</tr>
<tr>
<td>A.8 Move To/From Time Base Mnemonics</td>
<td>912</td>
</tr>
<tr>
<td>A.9 Return From Event-Based Branch Mnemonic</td>
<td>912</td>
</tr>
</tbody>
</table>

# Appendix B. Programming Examples for Sharing Storage

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>B.1 Atomic Update Primitives</td>
<td>913</td>
</tr>
<tr>
<td>B.2 Lock Acquisition and Release, and Related Techniques</td>
<td>915</td>
</tr>
<tr>
<td>B.2.1 Lock Acquisition and Import Barriers</td>
<td>915</td>
</tr>
<tr>
<td>B.2.1.1 Acquire Lock and Import Shared Storage</td>
<td>915</td>
</tr>
<tr>
<td>B.2.1.2 Obtain Pointer and Import Shared Storage</td>
<td>915</td>
</tr>
<tr>
<td>B.2.2 Lock Release and Export Barriers</td>
<td>916</td>
</tr>
<tr>
<td>B.2.2.1 Export Shared Storage and Release Lock</td>
<td>916</td>
</tr>
<tr>
<td>B.2.2.2 Export Shared Storage and Release Lock using lwsync</td>
<td>916</td>
</tr>
<tr>
<td>B.2.3 Safe Fetch</td>
<td>916</td>
</tr>
<tr>
<td>B.3 List Insertion</td>
<td>917</td>
</tr>
<tr>
<td>B.4 Notes</td>
<td>917</td>
</tr>
<tr>
<td>B.5 Transactional Lock Elision</td>
<td>917</td>
</tr>
<tr>
<td>B.5.1 Enter Critical Section</td>
<td>918</td>
</tr>
<tr>
<td>B.5.2 Handling Busy Lock</td>
<td>918</td>
</tr>
<tr>
<td>B.5.3 Handling TLE Abort</td>
<td>918</td>
</tr>
<tr>
<td>B.5.4 TLE Exit Section Critical Path</td>
<td>918</td>
</tr>
<tr>
<td>B.5.5 Acquisition and Release of TLE Locks</td>
<td>918</td>
</tr>
</tbody>
</table>

# Chapter 2. Logical Partitioning (LPAR) and Thread Control

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 Overview</td>
<td>927</td>
</tr>
<tr>
<td>2.2 Logical Partitioning Control Register (LPCR)</td>
<td>927</td>
</tr>
<tr>
<td>2.3 Hypervisor Real Mode Offset Register (HRMOR)</td>
<td>931</td>
</tr>
<tr>
<td>2.4 Logical Partition Identification Register (LPIDR)</td>
<td>931</td>
</tr>
<tr>
<td>2.5 Processor Compatibility Register (PCR)</td>
<td>932</td>
</tr>
<tr>
<td>2.6 Other Hypervisor Resources</td>
<td>941</td>
</tr>
<tr>
<td>2.7 Sharing Hypervisor Resources</td>
<td>941</td>
</tr>
<tr>
<td>2.8 Sub-Processors</td>
<td>942</td>
</tr>
<tr>
<td>2.9 Thread Identification Register (TIR)</td>
<td>942</td>
</tr>
<tr>
<td>2.10 Hypervisor Interrupt Little-Endian (HILE) Bit</td>
<td>942</td>
</tr>
</tbody>
</table>

# Chapter 3. Branch Facility

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 Branch Facility Overview</td>
<td>943</td>
</tr>
<tr>
<td>3.2 Branch Facility Registers</td>
<td>943</td>
</tr>
<tr>
<td>3.2.1 Machine State Register</td>
<td>943</td>
</tr>
<tr>
<td>3.2.2 State Transitions Associated with the Transactional Memory Facility</td>
<td>946</td>
</tr>
<tr>
<td>3.2.3 Processor Stop Status and Control Register (PSSCR)</td>
<td>949</td>
</tr>
<tr>
<td>3.3 Branch Facility Instructions</td>
<td>952</td>
</tr>
<tr>
<td>3.3.1 System Linkage Instructions</td>
<td>952</td>
</tr>
<tr>
<td>3.3.2 Power-Saving Mode</td>
<td>957</td>
</tr>
<tr>
<td>3.3.2.1 Power-Saving Mode Instruction</td>
<td>958</td>
</tr>
<tr>
<td>3.3.2.2 Entering and Exiting Power-Saving Mode</td>
<td>958</td>
</tr>
<tr>
<td>3.4 Event-Based Branch Facility and Instruction</td>
<td>960</td>
</tr>
</tbody>
</table>

# Chapter 4. Fixed-Point Facility

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1 Fixed-Point Facility Overview</td>
<td>961</td>
</tr>
<tr>
<td>4.2 Special Purpose Registers</td>
<td>961</td>
</tr>
<tr>
<td>4.3 Fixed-Point Facility Registers</td>
<td>961</td>
</tr>
<tr>
<td>4.3.1 Processor Version Register</td>
<td>961</td>
</tr>
<tr>
<td>4.3.2 Chip Information Register</td>
<td>961</td>
</tr>
<tr>
<td>4.3.3 Processor Identification Register</td>
<td>961</td>
</tr>
<tr>
<td>4.3.4 Process Identification Register</td>
<td>962</td>
</tr>
<tr>
<td>4.3.5 Thread ID Register</td>
<td>962</td>
</tr>
<tr>
<td>4.3.6 Control Register</td>
<td>962</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>4.3.7 Program Priority Register</td>
<td>963</td>
</tr>
<tr>
<td>4.3.8 Problem State Priority Boost Register</td>
<td>963</td>
</tr>
<tr>
<td>4.3.9 Relative Priority Register</td>
<td>963</td>
</tr>
<tr>
<td>4.3.10 Software-use SPRs</td>
<td>964</td>
</tr>
<tr>
<td>4.4 Fixed-Point Facility Instructions</td>
<td>965</td>
</tr>
<tr>
<td>4.4.1 Fixed-Point Load and Store Caching Inhibited Instructions</td>
<td>965</td>
</tr>
<tr>
<td>4.4.2 OR Instruction</td>
<td>968</td>
</tr>
<tr>
<td>4.4.3 Transactional Memory Instructions</td>
<td>969</td>
</tr>
<tr>
<td>4.4.4 Move To/From System Register Instructions</td>
<td>970</td>
</tr>
<tr>
<td>Chapter 5. Storage Control</td>
<td>981</td>
</tr>
<tr>
<td>5.1 Overview</td>
<td>981</td>
</tr>
<tr>
<td>5.2 Storage Exceptions</td>
<td>981</td>
</tr>
<tr>
<td>5.3 Instruction Fetch</td>
<td>981</td>
</tr>
<tr>
<td>5.3.1 Implicit Branch</td>
<td>981</td>
</tr>
<tr>
<td>5.3.2 Address Wrapping Combined with Changing MSR Bit SF</td>
<td>981</td>
</tr>
<tr>
<td>5.4 Data Access</td>
<td>982</td>
</tr>
<tr>
<td>5.5 Performing Operations</td>
<td>982</td>
</tr>
<tr>
<td>5.6 Out-of-Order Access</td>
<td>982</td>
</tr>
<tr>
<td>5.7 Invalid Real Address</td>
<td>982</td>
</tr>
<tr>
<td>5.7.1 32-Bit Mode</td>
<td>983</td>
</tr>
<tr>
<td>5.7.2 Virtualized Partition Memory (VPM) Mode</td>
<td>984</td>
</tr>
<tr>
<td>5.7.3 Hypervisor Real And Virtual Real Addressing Modes</td>
<td>984</td>
</tr>
<tr>
<td>5.7.3.1 Hypervisor Offset Real Mode Address</td>
<td>984</td>
</tr>
<tr>
<td>5.7.3.2 Storage Control Attributes for Accesses in Hypervisor Real Addressing Mode</td>
<td>984</td>
</tr>
<tr>
<td>5.7.3.2.1 Hypervisor Real Mode Storage Control</td>
<td>985</td>
</tr>
<tr>
<td>5.7.3.3 Virtual Real Mode Addressing Mechanism</td>
<td>985</td>
</tr>
<tr>
<td>5.7.3.4 Storage Control Attributes for Implicit Storage Accesses</td>
<td>986</td>
</tr>
<tr>
<td>5.7.4 Definitions</td>
<td>986</td>
</tr>
<tr>
<td>5.7.5 Address Ranges Having Defined Uses</td>
<td>987</td>
</tr>
<tr>
<td>5.7.5.1 Effective Address Space Structure for Radix-using Partitions</td>
<td>987</td>
</tr>
<tr>
<td>5.7.6 In-Memory Tables</td>
<td>988</td>
</tr>
<tr>
<td>5.7.6.1 Partition Table</td>
<td>989</td>
</tr>
<tr>
<td>5.7.6.2 Process Table</td>
<td>991</td>
</tr>
<tr>
<td>5.7.7 Address Translation Overview</td>
<td>991</td>
</tr>
<tr>
<td>5.7.8 Segment Translation</td>
<td>994</td>
</tr>
<tr>
<td>5.7.8.1 Segment Lookaside Buffer (SLB)</td>
<td>994</td>
</tr>
<tr>
<td>5.7.8.2 SLB Search</td>
<td>995</td>
</tr>
<tr>
<td>5.7.8.3 Segment Table Description and Search</td>
<td>995</td>
</tr>
<tr>
<td>5.7.8.3.1 Primary Hash for 256MB Segment</td>
<td>996</td>
</tr>
<tr>
<td>5.7.8.3.2 Primary Hash for 1TB Segment</td>
<td>996</td>
</tr>
<tr>
<td>5.7.8.3.3 Secondary Hash for 256MB Segment</td>
<td>996</td>
</tr>
<tr>
<td>5.7.8.3.4 Secondary Hash for 1TB Segment</td>
<td>996</td>
</tr>
<tr>
<td>5.7.9 Hashed Page Table Translation</td>
<td>996</td>
</tr>
<tr>
<td>5.7.9.1 Hashed Page Table</td>
<td>998</td>
</tr>
<tr>
<td>5.7.9.2 Page Table Search</td>
<td>999</td>
</tr>
<tr>
<td>5.7.10 Radix Tree Translation</td>
<td>1001</td>
</tr>
<tr>
<td>5.7.10.1 Radix Tree Page Directory Entry</td>
<td>1002</td>
</tr>
<tr>
<td>5.7.10.2 Radix Tree Page Table Entry</td>
<td>1003</td>
</tr>
<tr>
<td>5.7.10.3 Nested Translation</td>
<td>1003</td>
</tr>
<tr>
<td>5.7.11 Translation Process</td>
<td>1005</td>
</tr>
<tr>
<td>5.7.11.1 Fully-Qualified Address</td>
<td>1005</td>
</tr>
<tr>
<td>5.7.11.2 Finding the Page Tables</td>
<td>1006</td>
</tr>
<tr>
<td>5.7.11.3 Obtaining Host Real Address, Radix on Radix</td>
<td>1006</td>
</tr>
<tr>
<td>5.7.11.4 Obtaining Host Real Address, HPT</td>
<td>1007</td>
</tr>
<tr>
<td>5.7.12 Reference and Change Recording</td>
<td>1007</td>
</tr>
<tr>
<td>5.7.13 Storage Protection</td>
<td>1011</td>
</tr>
<tr>
<td>5.7.13.1 Virtual Page Class Key Protection</td>
<td>1011</td>
</tr>
<tr>
<td>5.7.13.2 Basic Storage Protection, Address Translation Enabled</td>
<td>1015</td>
</tr>
<tr>
<td>5.7.13.3 Basic Storage Protection, Address Translation Disabled</td>
<td>1016</td>
</tr>
<tr>
<td>5.7.13.4 Radix Tree Translation Storage Protection</td>
<td>1016</td>
</tr>
<tr>
<td>5.8 Storage Control Attributes</td>
<td>1017</td>
</tr>
<tr>
<td>5.8.1 Guarded Storage</td>
<td>1017</td>
</tr>
<tr>
<td>5.8.1.1 Out-of-Order Accesses to Guarded Storage</td>
<td>1018</td>
</tr>
<tr>
<td>5.8.2 Storage Control Bits</td>
<td>1018</td>
</tr>
<tr>
<td>5.8.2.1 Storage Control Bit Restrictions</td>
<td>1019</td>
</tr>
<tr>
<td>5.8.2.2 Altering the Storage Control Bits</td>
<td>1019</td>
</tr>
<tr>
<td>5.9 Storage Control Instructions</td>
<td>1021</td>
</tr>
<tr>
<td>5.9.1 Cache Management Instructions</td>
<td>1021</td>
</tr>
<tr>
<td>5.9.2 Synchronize Instruction</td>
<td>1021</td>
</tr>
<tr>
<td>5.9.3 Lookaside Buffer Management</td>
<td>1022</td>
</tr>
<tr>
<td>5.9.3.1 Thread-Specific Segment Translations</td>
<td>1023</td>
</tr>
<tr>
<td>5.9.3.2 SLB Management Instructions</td>
<td>1023</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.

\[
\begin{align*}
\text{stw} & \quad \text{RS},D(\text{RA}) \\
\text{addis} & \quad \text{RT},\text{RA},\text{SI}
\end{align*}
\]

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 in Appendix C of Book I.

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.

- **quadword, doubleword, word, halfword, and byte**
  128 bits, 64 bits, 32 bits, 16 bits, and 8 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 1.9.1 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 29.

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 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 Xp 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.
  - Xp means bit p of register/instruction/field/bit_string X.
  - Xpq means bits p through q of register/instruction/field/bit_string X.
  - Xpq... 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.

- xn means x raised to the nth power.
- nx means the replication of x, n times (i.e., x concatenated to itself n-1 times). n0 and n1 are special cases:
  - n0 means a field of n bits with each bit equal to 0. Thus n0 is equivalent to 0b00000.
  - n1 means a field of n bits with each bit equal to 1. Thus n1 is equivalent to 0b11111.

- Each bit and field in instructions, and in status and control registers (e.g., XER, FPSCR) and Special Purpose Registers, is either defined or reserved. Some defined fields contain reserved values. In such cases when this document refers to the specific field, it refers only to the defined values, unless otherwise specified.
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 1.9.2 on page 23. The only exception to the preceding rule is that it does not apply to Reserved and Illegal classes of instructions (see Section 1.8) 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 6 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 ((a = b) = (a ⊕ ¬b))
- Absolute value of 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”.
- Least integer ≥ x
- Result of converting x from floating-point single format to floating-point double format, using the model shown on page 140
- 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”.
- Result of extending x on the left with sign bits
- Greatest integer ≤ x
- General Purpose Register x
- Mask having 1s in positions x through y (wrapping if x > y) and 0s elsewhere
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.

ROTL64(x, y) Result of rotating the 64-bit value x left y positions
ROTL32(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 144
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. Does not correspond to any architected register. The CIA is sometimes referred to as the Program Counter (PC).

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.

for For loop, indenting shows range. Clause after “for” specifies the entities for which to execute the body of the loop.

switch/case/default switch/case/default statement, indenting shows range. The clause after “switch” specifies the expression to evaluate. The clause after “case” specifies individual values for the expression, followed by a colon, followed by the actions that are taken if the evaluated expression has any of the specified values. “default” is optional. If present, it must follow all the “case” clauses. The clause after “default” starts with a colon, and specifies the actions that are taken if the evaluated expression does not have any of the values specified in the preceding case statements.
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, \(-\) 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), post-superscript (exponentiation)</td>
<td>right to left</td>
</tr>
<tr>
<td>unary (-), (\neg)</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>(=), (&lt;), (\leq), (\geq), (&lt;u), (&gt;u), (?)</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>(\leftrightarrow), (\leftrightarrow)</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.

---

**Programming Note**

*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, half-word, word, doubleword, and quadword operands, where each operand contained in a GPR. Vector fixed-point instructions operate on vectors of byte, half-word, and word operands, where each vector is contained in a VR. Scalar floating-point instructions operate on single-precision or double-precision floating-point operands, where each operand is contained in an FPR or VSR. 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 bytes long and 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 word and doubleword operand loads and stores between storage and a set of 32 Floating-Point Registers (FPRs). It also provides for byte, halfword, word, and quadword operand loads and stores between storage and a set of 32 Vector Registers (VRs). It provides for doubleword and quadword 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.)
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,
Link Register, Count Register, etc.) and produce a 64-bit result. However, in 32-bit mode the high-order 32 bits of the computed effective address are ignored for the purpose of addressing storage; see Section 1.11.3 for additional details.

**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.

### 1.6 Instruction Formats

All instructions are four bytes long and word-aligned. Thus, 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.

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

The format diagrams given below show horizontally all valid combinations of instruction fields. The diagrams include instruction fields that are used only by instructions defined in Book II or in Book III.

**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 **A-FORM**

<table>
<thead>
<tr>
<th>0</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>26</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>PO</td>
<td>FRT ///</td>
<td>FRB ///</td>
<td>XO</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PO</td>
<td>FRT FRA ///</td>
<td>FRC XO</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PO</td>
<td>FRT FRA FRB ///</td>
<td>XO</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PO</td>
<td>RT RA RB BC XO</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 3. A instruction format

1.6.2 **B-FORM**

<table>
<thead>
<tr>
<th>0</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>3031</th>
</tr>
</thead>
<tbody>
<tr>
<td>PO</td>
<td>BO</td>
<td>BI</td>
<td>BD</td>
<td></td>
</tr>
</tbody>
</table>

Figure 4. B instruction format

1.6.3 **D-FORM**

<table>
<thead>
<tr>
<th>0</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>PO</td>
<td>BF ///</td>
<td>LRA</td>
<td>SI</td>
<td></td>
</tr>
<tr>
<td>PO</td>
<td>RFS RA</td>
<td>D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PO</td>
<td>FRT RA</td>
<td>D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PO</td>
<td>RS RA</td>
<td>D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PO</td>
<td>RT RA</td>
<td>D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PO</td>
<td>TO RA</td>
<td>SI</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 5. D instruction format

1.6.4 **DQ-FORM**

<table>
<thead>
<tr>
<th>0</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>2829</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>PO</td>
<td>RTP</td>
<td>RA</td>
<td>DQ</td>
<td>PT</td>
<td></td>
</tr>
<tr>
<td>PO</td>
<td>S</td>
<td>RA</td>
<td>DQ</td>
<td>SI</td>
<td>XO</td>
</tr>
<tr>
<td>PO</td>
<td>T</td>
<td>RA</td>
<td>DQ</td>
<td>SI</td>
<td>XO</td>
</tr>
</tbody>
</table>

Figure 6. DQ instruction format

1.6.5 **DS-FORM**

<table>
<thead>
<tr>
<th>0</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>3031</th>
</tr>
</thead>
<tbody>
<tr>
<td>PO</td>
<td>FRSp</td>
<td>RA</td>
<td>DS</td>
<td>XO</td>
</tr>
<tr>
<td>PO</td>
<td>FRTp</td>
<td>RA</td>
<td>DS</td>
<td>XO</td>
</tr>
<tr>
<td>PO</td>
<td>RS</td>
<td>RA</td>
<td>DS</td>
<td>XO</td>
</tr>
<tr>
<td>PO</td>
<td>RSp</td>
<td>RA</td>
<td>DS</td>
<td>XO</td>
</tr>
<tr>
<td>PO</td>
<td>VRS</td>
<td>RA</td>
<td>DS</td>
<td>XO</td>
</tr>
<tr>
<td>PO</td>
<td>VRT</td>
<td>RA</td>
<td>DS</td>
<td>XO</td>
</tr>
</tbody>
</table>

Figure 7. DS instruction format

1.6.6 **DX-FORM**

<table>
<thead>
<tr>
<th>0</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>26</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>PO</td>
<td>RT</td>
<td>d1</td>
<td>d0</td>
<td>XO</td>
<td></td>
</tr>
</tbody>
</table>

Figure 8. DX instruction format

1.6.7 **I-FORM**

<table>
<thead>
<tr>
<th>0</th>
<th>6</th>
<th>3031</th>
</tr>
</thead>
<tbody>
<tr>
<td>PO</td>
<td>LI</td>
<td></td>
</tr>
</tbody>
</table>

Figure 9. I instruction format

1.6.8 **M-FORM**

<table>
<thead>
<tr>
<th>0</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>26</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>PO</td>
<td>RS</td>
<td>RA</td>
<td>RB</td>
<td>MB</td>
<td>ME</td>
<td></td>
</tr>
<tr>
<td>PO</td>
<td>RS</td>
<td>RA</td>
<td>SH</td>
<td>MB</td>
<td>ME</td>
<td></td>
</tr>
</tbody>
</table>

Figure 10. M instruction format

1.6.9 **MD-FORM**

<table>
<thead>
<tr>
<th>0</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>27</th>
<th>3031</th>
</tr>
</thead>
<tbody>
<tr>
<td>PO</td>
<td>RS</td>
<td>RA</td>
<td>sh</td>
<td>mb</td>
<td>XO</td>
<td>a</td>
</tr>
<tr>
<td>PO</td>
<td>RS</td>
<td>RA</td>
<td>sh</td>
<td>me</td>
<td>XO</td>
<td>a</td>
</tr>
</tbody>
</table>

Figure 11. MD instruction format

1.6.10 **MDS-FORM**

<table>
<thead>
<tr>
<th>0</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>25</th>
<th>27</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>PO</td>
<td>RS</td>
<td>RA</td>
<td>RB</td>
<td>MB</td>
<td>ME</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PO</td>
<td>RS</td>
<td>RA</td>
<td>RB</td>
<td>ME</td>
<td>XO</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 12. MDS instruction format

1.6.11 **SC-FORM**

<table>
<thead>
<tr>
<th>0</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>20</th>
<th>27</th>
<th>3031</th>
</tr>
</thead>
<tbody>
<tr>
<td>PO /// /// /// LEV /// 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 13. SC instruction format

1.6.12 **VA-FORM**

<table>
<thead>
<tr>
<th>0</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>2122</th>
<th>26</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>PO</td>
<td>RT</td>
<td>RA</td>
<td>RB</td>
<td>RC</td>
<td>XO</td>
<td></td>
</tr>
<tr>
<td>PO</td>
<td>VRT</td>
<td>VRA</td>
<td>VRB</td>
<td>SHB</td>
<td>XO</td>
<td></td>
</tr>
<tr>
<td>PO</td>
<td>VRT</td>
<td>VRA</td>
<td>VRB</td>
<td>VRC</td>
<td>XO</td>
<td></td>
</tr>
</tbody>
</table>

Figure 14. VA instruction format

1.6.13 **VC-FORM**

<table>
<thead>
<tr>
<th>0</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>2122</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>PO</td>
<td>VRT</td>
<td>VRA</td>
<td>VRB</td>
<td>SHB</td>
<td>XO</td>
</tr>
</tbody>
</table>

Figure 15. VC instruction format
1.6.14 VX-FORM

Figure 16. VX instruction format

1.6.15 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 | 1 |
| PO | RS | RA | RB | XO | 1 |
| PO | RS | RA | RB | XO | 1 |
| PO | RT | // | // | XO | / |
| PO | RT | // | // | XO | / |
| PO | RT | // | // | XO | 1 |
| PO | RT | // | // | XO | / |
| PO | RT | // | // | XO | / |
| PO | RT | // | // | XO | / |
| PO | RT | // | // | XO | / |
| PO | RT | // | // | XO | / |
| PO | RT | // | // | XO | / |
| PO | RT | // | // | XO | / |
| PO | RT | // | // | XO | / |
| PO | RT | // | // | XO | / |
| PO | RT | // | // | XO | / |
| PO | S | RA | /// | XO | / |
| PO | S | RA | /// | XO | / |
| PO | T | EO | IMM8 | XO | / |
| PO | T | RA | /// | XO | / |
| PO | T | RA | RB | XO | / |
| PO | TO | RA | SI | XO | 1 |
| PO | TO | RA | RB | XO | / |
| PO | TO | RA | RB | XO | / |
| PO | VRS | RA | RB | XO | / |
| PO | VRT | EO | VRB | XO | / |
| PO | VRT | EO | VRB | XO | / |
| PO | VRT | RA | RB | XO | / |
| PO | VRT | VRA | VRB | XO | / |
| PO | VRT | VRA | VRB | XO | / |
| PO | VRT | VRA | VRB | XO | / |

Figure 17. X instruction format
Chapter 1. Introduction

1.6.16  **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>FL</td>
<td>M</td>
<td>FRB</td>
<td>XO</td>
</tr>
</tbody>
</table>

**Figure 18. XFL instruction format**

1.6.17  **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>FX</td>
<td>M</td>
<td>XO</td>
</tr>
<tr>
<td>PO</td>
<td>RS</td>
<td>1</td>
<td>FX</td>
<td>M</td>
<td>XO</td>
</tr>
<tr>
<td>PO</td>
<td>RT</td>
<td>0</td>
<td>spr</td>
<td>XO</td>
<td></td>
</tr>
<tr>
<td>PO</td>
<td>RT</td>
<td>1</td>
<td>FX</td>
<td>M</td>
<td>XO</td>
</tr>
<tr>
<td>PO</td>
<td>RT</td>
<td>BHRBE</td>
<td>XO</td>
<td></td>
<td></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**

1.6.18  **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>///</td>
<td>XO</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PO</td>
<td>BF</td>
<td>///</td>
<td>BFA</td>
<td>///</td>
<td>///</td>
<td>XO</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**

1.6.19  **XO-FORM**

<table>
<thead>
<tr>
<th>0</th>
<th>6</th>
<th>9</th>
<th>10111213141516</th>
<th>171819202122232425262728293031</th>
</tr>
</thead>
<tbody>
<tr>
<td>PO</td>
<td>RT</td>
<td>RA</td>
<td>///</td>
<td>XO</td>
</tr>
<tr>
<td>PO</td>
<td>RT</td>
<td>RA</td>
<td>RB</td>
<td>XO</td>
</tr>
<tr>
<td>PO</td>
<td>RT</td>
<td>RA</td>
<td>RB</td>
<td>XO</td>
</tr>
<tr>
<td>PO</td>
<td>RT</td>
<td>RA</td>
<td>RB</td>
<td>XO</td>
</tr>
</tbody>
</table>

**Figure 21. XO instruction format**

1.6.20  **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>a</td>
</tr>
</tbody>
</table>

**Figure 22. XS instruction format**

1.6.21  **XX2-FORM**

<table>
<thead>
<tr>
<th>0</th>
<th>6</th>
<th>9</th>
<th>10111213141516</th>
<th>21</th>
<th>2526</th>
<th>293031</th>
</tr>
</thead>
<tbody>
<tr>
<td>PO</td>
<td>BF</td>
<td>///</td>
<td>///</td>
<td>B</td>
<td>XO</td>
<td>31</td>
</tr>
<tr>
<td>PO</td>
<td>BF</td>
<td>DCMX</td>
<td>B</td>
<td>XO</td>
<td>31</td>
<td></td>
</tr>
<tr>
<td>PO</td>
<td>RT</td>
<td>EO</td>
<td>B</td>
<td>XO</td>
<td>31</td>
<td></td>
</tr>
<tr>
<td>PO</td>
<td>T</td>
<td>///</td>
<td>B</td>
<td>XO</td>
<td>31</td>
<td></td>
</tr>
<tr>
<td>PO</td>
<td>T</td>
<td>///</td>
<td>UIM</td>
<td>B</td>
<td>XO</td>
<td>31</td>
</tr>
<tr>
<td>PO</td>
<td>T</td>
<td>dx</td>
<td>B</td>
<td>XO</td>
<td>a</td>
<td>31</td>
</tr>
<tr>
<td>PO</td>
<td>T</td>
<td>EO</td>
<td>B</td>
<td>XO</td>
<td>31</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 23. XX2 instruction format**

1.6.22  **XX3-FORM**

<table>
<thead>
<tr>
<th>0</th>
<th>6</th>
<th>9</th>
<th>11</th>
<th>16</th>
<th>212224</th>
<th>293031</th>
</tr>
</thead>
<tbody>
<tr>
<td>PO</td>
<td>BF</td>
<td>///</td>
<td>A</td>
<td>B</td>
<td>XO</td>
<td>a</td>
</tr>
<tr>
<td>PO</td>
<td>T</td>
<td>A</td>
<td>B</td>
<td>0</td>
<td>sh</td>
<td>XO</td>
</tr>
<tr>
<td>PO</td>
<td>T</td>
<td>A</td>
<td>B</td>
<td>0</td>
<td>sh</td>
<td>XO</td>
</tr>
<tr>
<td>PO</td>
<td>T</td>
<td>A</td>
<td>B</td>
<td>Rs</td>
<td>XO</td>
<td>a</td>
</tr>
<tr>
<td>PO</td>
<td>T</td>
<td>A</td>
<td>B</td>
<td>XO</td>
<td>a</td>
<td>31</td>
</tr>
</tbody>
</table>

**Figure 24. XX3 instruction format**

1.6.23  **XX4-FORM**

<table>
<thead>
<tr>
<th>0</th>
<th>6</th>
<th>9</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>
<td>31</td>
</tr>
</tbody>
</table>

**Figure 25. XX4 instruction format**

1.6.24  **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>FRA</td>
<td>DGM</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>DGM</td>
<td>XO</td>
<td></td>
</tr>
<tr>
<td>PO</td>
<td>BF</td>
<td>///</td>
<td>FRAp</td>
<td>SH</td>
<td>XO</td>
<td></td>
</tr>
<tr>
<td>PO</td>
<td>FR</td>
<td>FRAp</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**
1.7 Instruction Fields

A (6)
Field used by the `tbegin` instruction to specify an implementation-specific function.
Field used by the `tend` instruction to specify the completion of the outer transaction and all nested transactions.
Formats: X

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 0b0000 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 mtssf instruction.
Formats: XFL

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

FRAp (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 \texttt{mtcrf} and \texttt{mtocrf} instructions, or read by the \texttt{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 \texttt{SLB Invalidate All} instruction. The meaning is described in Section 5.9.3.2, “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 \texttt{mtfsf} instruction updates the entire FPSCR.
Formats: XFL

L (9:10)
Field used by the \textit{Data Cache Block Flush} instruction (see Section 4.3.2 of Book II) and also by the \textit{Synchronize} instruction (see Section 4.6.3 of Book II).
Formats: X

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 \textit{Compare Range Byte} instruction to indicate whether to compare against 1 or 2 ranges of bytes.
Formats: D, X

L (15)
Field used by the \textit{Move To Machine State Register} instruction (see Book III).
Field used by the \textit{SLB Move From Entry VSID} and \textit{SLB Move From Entry ESID} instructions for implementation-specific purposes.
Formats: X

L (14:15)
Field used by the \textit{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 \textit{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 0b000 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 \textit{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 101.

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 101.

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 101.

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 101.

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

PO (0:5)
Primary opcode.

Formats: all

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

Formats: X

PS (22)
Field used to specify preferred sign for BCD operations.

Formats: VX

PT (28:31)
Immediate field used to specify a 4-bit unsigned value.

Formats: DQ

R (10)
Field used by the tbegin, instruction to specify the start of a ROT.

Formats: X

R (15)
Immediate field that specifies whether the RMC is specifying the primary or secondary encoding

Field used to specify whether to invalidate Radix Tree or HPT entries for tlbie[f].

Formats: X, Z23

RA (11:15)
Field used to specify a GPR to be used as a source or as a target.

Formats: A, D, DQ, DQE, DS, M, MD, MDS, TX, VA, VX, X, XO, XS

RB (16:20)
Field used to specify a GPR to be used as a source.

Formats: A, M, MDS, VA, X, XO

Rc (21)
RECORD bit.

0 Do not alter the Condition Register.

1 Set Condition Register Field 6 as described in Section 2.3.1, “Condition Register” on page 30.

Formats: VC, XX3

RC (21:25)
Field used to specify a GPR to be used as a source.

Formats: VA

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 30.

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[f].

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

RO (31)
Round to Odd override
Formats: X

RS (6:10)
Field used to specify a GPR to be used as a source.
Formats: D, DS, M, MD, MDS, X, XFX, XS

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

RT (6:10)
Field used to specify a GPR to be used as a target.
Formats: A, D, DQE, DS, DX, VA, VX, X, XFX, XO, XX2

RTp (6:10)
Field used to specify an even/odd pair of GPRs to be concatenated and used as a target.
Formats: DQ, X

S (11)
Immediate field that specifies signed versus unsigned conversion.
Formats: X

S (20)
Immediate field that specifies whether or not the reebb instruction re-enables event-based branches.
Formats: XL

SH (16:20)
Field used to specify a shift amount.
Formats: M, X

SH (16:21)
Field used to specify a shift amount.
Formats: Z22

sh (30,16:20)
Fields that are concatenated to specify a shift amount.
Formats: MD, XS

SHB (22:25)
Field used to specify a shift amount in bytes.
Formats: VA

SHW (22:23)
Field used to specify a shift amount in words.
Formats: X

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

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

SIM (11:15)
Immediate field used to specify a 5-bit signed integer.
Formats: VX

SP (11:12)
Immediate field that specifies signed versus unsigned conversion.
Formats: X

SPR (11:20)
Field used to specify a Special Purpose Register for the mtspr and mfspr instructions.
Formats: X

SR (12:15)
Field used by the Segment Register Manipulation instructions (see Book III).
Formats: X

SX,S (28,6:10)
Fields SX and S are concatenated to specify a VSR to be used as a source.
Formats: DQ

SX,S (31,6:10)
Fields SX and S are concatenated to specify a VSR to be used as a source.
Formats: X

TBR (11:20)
Field used by the Move From Time Base instruction (see Section 6.1 of Book II).
Formats: X

TE (11:15)
Immediate field that specifies a DFP exponent.
Formats: Z23

TH (6:10)
Field used by the data stream variant of the dcbt and dcbtsft instructions (see Section 4.3.2 of Book II).
Formats: X
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 87.
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
1.8 Classes of Instructions

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

1.8.1 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 1.9.1, “Preferred Instruction Forms” and Section 1.9.2, “Invalid Instruction Forms”.

1.8.2 Illegal Instruction Class

This class of instructions contains the set of instructions described in Appendix A 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.

1.8.3 Reserved Instruction Class

This class of instructions contains the set of instructions described in Appendix B 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.

---

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

XO (22:30)
Extended opcode field.
Formats: XO, XX3, Z22

XO (22:31)
Extended opcode field.
Formats: VC

XO (23:30)
Extended opcode field.
Formats: X, Z23

XO (25:30)
Extended opcode field.
Formats: TX

XO (26:27)
Extended opcode field.
Formats: XX4

XO (26:30)
Extended opcode field.
Formats: A, DX

XO (26:31)
Extended opcode field.
Formats: VA

XO (27:29)
Extended opcode field.
Formats: MD

XO (27:30)
Extended opcode field.
Formats: MDS

XO (29:31)
Extended opcode field.
Formats: DQ

XO (30)
Extended opcode field.
Formats: SC

XO (30:31)
Extended opcode field.
Formats: DQE, DS, SC

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.
1.9 Forms of Defined Instructions

1.9.1 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

1.9.2 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.

<table>
<thead>
<tr>
<th>Assembler Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assemblers should report uses of invalid instruction forms as errors.</td>
</tr>
</tbody>
</table>

1.9.3 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.

1.10 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 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 Vectored 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 execution of an auxiliary processor instruction that causes an auxiliary processor enabled exception to exist (system auxiliary processor 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 the invocation of the auxiliary processor enabled exception error handler may be imprecise, and
if one of the imprecise modes for invoking the system floating-point enabled exception error handler is in effect (see page 133), 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.

1.11 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 3.2.1 of Book III), which applies to all of storage.

1.11.1 Storage Operands

A storage operand may be a byte, a halfword, a word, a doubleword, or a quadword, or, for the Load/Store Multiple and Move Assist instructions, a sequence of bytes (Move Assist) 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, and quadword.

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), and the length of a Store Quadword, Store Floating-Point Double Pair, and Store VSX Vector Word*4 instruction is 16 bytes (one quadword). 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>Addr60:63 if aligned</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte</td>
<td>8 bits</td>
<td>xxxxx</td>
</tr>
<tr>
<td>Halfword</td>
<td>2 bytes</td>
<td>xxx0</td>
</tr>
<tr>
<td>Word</td>
<td>4 bytes</td>
<td>xxx0</td>
</tr>
<tr>
<td>Doubleword</td>
<td>8 bytes</td>
<td>x000</td>
</tr>
<tr>
<td>Quadword</td>
<td>16 bytes</td>
<td>0000</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.

![Big-Endian Byte Ordering](image)

<table>
<thead>
<tr>
<th>Load</th>
<th>Store</th>
</tr>
</thead>
<tbody>
<tr>
<td>for i=0 to N-1:</td>
<td>for i=0 to N-1:</td>
</tr>
<tr>
<td>RT&lt;sub&gt;(R-N)+i&lt;/sub&gt; = MEM(EA+i,1)</td>
<td>MEM(EA+i,1) = (RS)&lt;sub&gt;(R-N)+i&lt;/sub&gt;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Little-Endian Byte Ordering</th>
</tr>
</thead>
<tbody>
<tr>
<td>for i=0 to N-1:</td>
</tr>
<tr>
<td>RT&lt;sub&gt;(R-1)-i&lt;/sub&gt; = MEM(EA+i,1)</td>
</tr>
</tbody>
</table>

**Notes:**
1. In this table, subscripts refer to bytes in a register rather than to bits as defined in Section 1.3.2.
2. This table does not apply to the *lvebx*, *lvehx*, *lvewx*, *stvebx*, *stvehx*, and *stvewx* instructions.

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.

![C structure 's', showing values of elements](image)

```c
struct {
    int a; /* 0x1112_1314 word */
    double b; /* 0x2122_2324_2526_2728 doubleword */
    char * c; /* 0x3132_3334 word */
    char d[7]; /* 'A', 'B', 'C', 'D', 'E', 'F', 'G' array of bytes */
    short e; /* 0x5152 halfword */
    int f; /* 0x6162_6364 word */
} s;
```

![Big-Endian mapping of structure 's'](image)

![Little-Endian mapping of structure 's'](image)
1.11.2 Instruction Fetches

Instructions are always four bytes long and word-aligned.

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

![Figure 32. Instructions and byte ordering](image)

Figure 32 shows an example of a small assembly language program p.

```
loop:
  cmplwi r5,0
  beq done
  lwzux r4,r5,r6
  add r7,r7,r4
  subi r5,r5,4
  b loop

done:
  stw r7,total
```

![Figure 33. Assembly language program 'p'](image)

Figure 33. Assembly language program 'p'

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

![Figure 34. Big-Endian mapping of program 'p'](image)

Figure 34. Big-Endian mapping of program 'p'

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

![Figure 35. Little-Endian mapping of program 'p'](image)

Figure 35. Little-Endian mapping of program 'p'
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 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 Brundreca, (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 Seamen 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.

### 1.11.3 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 each 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 at effective address $2^{64} - 4$ the effective address of the next sequential instruction 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 at effective address $2^{32} - 4$ the 64-bit effective address of the next sequential instruction is undefined. Thus, as used to address storage, the effective address arithmetic appears to wrap around from the maximum address $2^{32} - 1$, 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 the instruction at effective address \(2^{32} - 4\) 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 lswi and stswi) 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 D-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 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 DQ-form instructions, 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 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 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, the value 4 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 (\(2^{34} - 4\) in 64-bit mode, \(2^{32} - 4\) 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.

- Transaction failure will eventually cause the transaction's failure handler, implied by the **tbegin** instruction, to be invoked. See the programming note following the **tbegin** description in Section 5.5 of Book II.

- Event-based exceptions can cause the event-based branch handler to be invoked, as described in Chapter 7 of Book II.

- Exceptions can cause the system error handler to be invoked, as described in Section 1.10, "Exceptions" on page 23.

- 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 1.9 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).

Figure 36. Condition Register

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 (mcrxx), 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 or of a tcheck instruction (see Book II).

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.

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></td>
<td>The result is negative.</td>
</tr>
<tr>
<td>1</td>
<td>Positive (GT)</td>
</tr>
<tr>
<td></td>
<td>The result is positive.</td>
</tr>
<tr>
<td>2</td>
<td>Zero (EQ)</td>
</tr>
<tr>
<td></td>
<td>The result is zero.</td>
</tr>
<tr>
<td>3</td>
<td>Summary Overflow (SO)</td>
</tr>
<tr>
<td></td>
<td>This is a copy of the contents of XERSO at the completion of the instruction.</td>
</tr>
</tbody>
</table>

With the exception of tcheck, the Transactional Memory instructions set CR002 indicating the state of the facility prior to instruction execution, or transaction failure. A complete description of the meaning of these bits is given in the instruction descriptions in Section 5.5 of Book II. These bits are interpreted as follows:

<table>
<thead>
<tr>
<th>CR0</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>Transaction state of Non-transactional prior to instruction</td>
</tr>
<tr>
<td>010</td>
<td>Transaction state of Transactional prior to instruction</td>
</tr>
<tr>
<td>001</td>
<td>Transaction state of Suspended prior to instruction</td>
</tr>
<tr>
<td>101</td>
<td>Transaction failure</td>
</tr>
</tbody>
</table>

The tcheck instruction similarly sets bits 1 and 2 of CR field BF to indicate the transaction state, and additionally sets bit 0 to TDOOMED, as defined in Section 5.5 of Book II.

<table>
<thead>
<tr>
<th>CR field BF</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TDOOMED</td>
<td></td>
</tr>
<tr>
<td>TDOOMED</td>
<td></td>
</tr>
<tr>
<td>TDOOMED</td>
<td></td>
</tr>
</tbody>
</table>

**Programming Note**

Setting of bit 3 of the specified CR field to zero by tcheck and of field CR03 to zero by other TM instructions is intended to preserve these bits for future function. Software should not depend on the bits being zero.
The *paste* instruction (see Section 4.4, “Copy-Paste Facility”, in Book II) and the *stbcx*, *stcx*, *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 132). 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 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 the completion of the instruction.</td>
</tr>
<tr>
<td>34</td>
<td>Floating-Point Invalid Operation Exception Summary (VX)</td>
</tr>
<tr>
<td></td>
<td>This is a copy of the contents of FPSCRVX at 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 FPSCROI at 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 84, and Section 4.6.8, “Floating-Point Compare Instructions” on page 167.

<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>
<tr>
<td></td>
<td>For fixed-point Compare instructions, (RA) &lt; SI or (RB) (signed comparison) or (RA) &lt;u UI or (RB) (unsigned comparison). For floating-point Compare instructions, (FRA) &lt; (FRB).</td>
</tr>
<tr>
<td>1</td>
<td>Greater Than, Floating-Point Greater Than (GT, FG)</td>
</tr>
<tr>
<td></td>
<td>For fixed-point Compare instructions, (RA) &gt; SI or (RB) (signed comparison) or (RA) &gt;u UI or (RB) (unsigned comparison). For floating-point Compare instructions, (FRA) &gt; (FRB).</td>
</tr>
<tr>
<td>2</td>
<td>Equal, Floating-Point Equal (EQ, FE)</td>
</tr>
<tr>
<td></td>
<td>For fixed-point Compare instructions, (RA) = SI, UI, or (RB). For floating-point Compare instructions, (FRA) = (FRB).</td>
</tr>
</tbody>
</table>

### 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>
<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 Compare Bounds Floating-Point instruction on page 328 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.
Bit  Description
0  0
1  0
2  Set to indicate whether all four elements in VRA are within the bounds specified by the corresponding element in VRB, otherwise set to 0.
3  0

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.

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.

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.

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 1.11.3, “Effective Address Calculation” on page 27.

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>0000z</td>
<td>Decrement the CTR, then branch if the decremented CTR_{63:0} \neq 0 and CR_{BI}=0</td>
</tr>
<tr>
<td>0001z</td>
<td>Decrement the CTR, then branch if the decremented CTR_{63:0} = 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_{63:0} \neq 0 and CR_{BI}=1</td>
</tr>
<tr>
<td>0101z</td>
<td>Decrement the CTR, then branch if the decremented CTR_{63:0} = 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_{63:0} \neq 0</td>
</tr>
<tr>
<td>1a01t</td>
<td>Decrement the CTR, then branch if the decremented CTR_{63:0} = 0</td>
</tr>
<tr>
<td>1zzz</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}[l]: The instruction is a subroutine return \textit{bcctr}[l] and \textit{bctar}[l]: 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{bclr}[l]: 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{bcctr}[l] and \textit{bctar}[l]: Reserved</td>
</tr>
<tr>
<td>10</td>
<td>Reserved</td>
</tr>
<tr>
<td>11</td>
<td>\textit{bclr}[l], \textit{bcctr}[l], and \textit{bctar}[l]: 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 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 bclr[l] and bcctr[l] instructions. These mechanisms may cache return addresses (i.e., Link Register values set by Branch instructions for which 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 Branch instructions for which 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 Branch instruction for which LK=1 and the branch is taken, other than the special form shown in the first example below) with a bclr instruction that returns from the subroutine and has BH=0b00.
- Do not use bclrl as a subroutine call. (Some implementations access the return address cache at most once per instruction; such implementations are likely to treat bclrl as a subroutine return, and not as a subroutine call.)
- For bclr[l] and 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 Branch and Link.
  \[ bcl \ 20,31,\$+4 \]

- Loop counts:
  Keep them in the Count Register, and use a 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 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 bl or bcl instruction (LK=1).
  - B returns to A: use a 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 bl or bcl instruction (LK=1).
  - Glue calls B: place the address of B into the Count Register, and use a bcctr instruction (LK=0).
  - B returns to A: use a 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 bcctr instruction (LK=1) instead of a bl or bcl instruction.
  - For the 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 $bcct[r][l]$), 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**

| b | target_addr | (AA=0 LK=0) |
| ba | target_addr | (AA=1 LK=0) |
| bl | target_addr | (AA=0 LK=1) |
| bla | target_addr | (AA=1 LK=1) |

**Branch Conditional B-form**

| bc | BO,BI,target_addr | (AA=0 LK=0) |
| bca | BO,BI,target_addr | (AA=1 LK=0) |
| bcl | BO,BI,target_addr | (AA=0 LK=1) |
| bcla | BO,BI,target_addr | (AA=1 LK=1) |

**Special Registers Altered:**

- LR (if LK=1)

**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.

**Extended Mnemonics:**

Examples of extended mnemonics for **Branch Conditional**:

- Extended: **blt target**
  - Equivalent to: **bc 12,0,target**
- Extended: **bne cr2,target**
  - Equivalent to: **bc 4,10,target**
- Extended: **bdnz target**
  - Equivalent to: **bc 16,0,target**
Branch Conditional to Link Register

**XL-form**

| bclr | BO, BI, BH | (LK=0) |
| bclrl | BO, BI, BH | (LK=1) |

<table>
<thead>
<tr>
<th>19</th>
<th>BO</th>
<th>BI</th>
<th>BH</th>
<th>16</th>
<th>LK</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>11</td>
<td>16</td>
<td>19</td>
<td>21</td>
<td>31</td>
</tr>
</tbody>
</table>

if (64-bit mode)
then M := 0
else M := 32
if ¬BO2 then CTR := CTR - 1

cond_ok := BO2 ∨ ((CTR[6:63] ≠ 0) ⊕ BO3)
if cond_ok & ¬BO2 then NIA := lea LR[0:61] || 0b00
if LK then LR := lea 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 LR[0:61] || 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)

**Extended Mnemonics:**

Examples of extended mnemonics for Branch Conditional to Link Register:

<table>
<thead>
<tr>
<th>Extended</th>
<th>Equivalent to</th>
</tr>
</thead>
<tbody>
<tr>
<td>bclr 4,6</td>
<td>bclr 4,6,0</td>
</tr>
<tr>
<td>blt 12,0,0</td>
<td>bclr 12,0,0</td>
</tr>
<tr>
<td>bnlr cr2</td>
<td>bclr 4,10,0</td>
</tr>
<tr>
<td>bdnzr 16,0</td>
<td>bclr 16,0,0</td>
</tr>
</tbody>
</table>

**Programming Note**

*bclr, bclrl, bcctr*, and *bcclrl* each serve as both a basic and an extended mnemonic. The Assembler will recognize a *bclr, bclrl, bcctr*, or *bcclrl* mnemonic with three operands as the basic form, and a *bcclrl, bclrl, bcctr*, or *bcclrl* 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**

| bcctr | BO, BI, BH | (LK=0) |
| bcclrl | BO, BI, BH | (LK=1) |

<table>
<thead>
<tr>
<th>19</th>
<th>BO</th>
<th>BI</th>
<th>BH</th>
<th>528</th>
<th>LK</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>11</td>
<td>16</td>
<td>19</td>
<td>21</td>
<td>31</td>
</tr>
</tbody>
</table>

cond_ok := BO0 ∨ (CRBI[32] = BO3)
if cond_ok then NIA := lea CTR[0:61] || 0b00
if LK then LR := lea 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 CTR[0:61] || 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</th>
<th>Equivalent to</th>
</tr>
</thead>
<tbody>
<tr>
<td>bcctr 4,6</td>
<td>bcctr 4,6,0</td>
</tr>
<tr>
<td>bltctr 12,0,0</td>
<td>bcctr 12,0,0</td>
</tr>
<tr>
<td>bnectr cr2</td>
<td>bcctr 4,10,0</td>
</tr>
</tbody>
</table>
**Branch Conditional to Branch Target Address Register**

**XL-form**

<table>
<thead>
<tr>
<th>BO</th>
<th>BL</th>
<th>BH</th>
<th>M</th>
<th>LK</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>11</td>
<td>16</td>
<td>19</td>
</tr>
</tbody>
</table>

if (64-bit mode)

    then M ← 0

else M ← 32

if ¬BO₂ then CTR ← CTR - 1

ctr_ok ← BO₂ | ((CTR₉:63 ≠ 0) ⊕ BO₃)

cond_ok ← BO₀ | (CRBI:32 = BO₄)

if ctr_ok & cond_ok then NIA ← IJE TAR₀:61 || 0b00

if LK then LR ← IJE 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 1.9.1. 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.

<table>
<thead>
<tr>
<th>Condition Register AND</th>
<th>XL-form</th>
<th>Condition Register NAND</th>
<th>XL-form</th>
</tr>
</thead>
<tbody>
<tr>
<td>crand</td>
<td>BT,BA,BB</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CRBT+32 ← CRBA+32 &amp; CRBB+32</td>
<td></td>
<td>CRBT+32 ← ¬(CRBA+32 &amp; CRBB+32)</td>
<td></td>
</tr>
<tr>
<td>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.</td>
<td>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 complemented result is placed into the bit in the Condition Register specified by BT+32.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Special Registers Altered:</td>
<td>CRBT+32</td>
<td>Special Registers Altered:</td>
<td>CRBT+32</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Condition Register OR</th>
<th>XL-form</th>
<th>Condition Register XOR</th>
<th>XL-form</th>
</tr>
</thead>
<tbody>
<tr>
<td>cror</td>
<td>BT,BA,BB</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CRBT+32 ← CRBA+32</td>
<td>CRBB+32</td>
<td>CRBT+32 ← CRBA+32 ⊕ CRBB+32</td>
<td></td>
</tr>
<tr>
<td>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.</td>
<td>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 result is placed into the bit in the Condition Register specified by BT+32.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Special Registers Altered:</td>
<td>CRBT+32</td>
<td>Special Registers Altered:</td>
<td>CRBT+32</td>
</tr>
</tbody>
</table>

Extended Mnemonics:

Example of extended mnemonics for Condition Register OR:

- Extended: crmove Bx,By
- Equivalent to: cror Bx,By,By

Example of extended mnemonics for Condition Register XOR:

- Extended: crclr Bx
- Equivalent to: crxor Bx,Bx,Bx
Chapter 2. Branch Facility

### Condition Register NOR

**XL-form**

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

\[
\begin{array}{cccccc}
0 & 19 & 6 & 8 & 11 & 16 & 21 & 23 & 33 & 31 \\
\end{array}
\]

\(\text{CR}_{\text{BT+32}} \leftarrow \neg (\text{CR}_{\text{BA+32}} \lor \neg \text{CR}_{\text{BB+32}})\)

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:**

\(\text{CR}_{\text{BT+32}}\)

**Extended Mnemonics:**

Example of extended mnemonics for **Condition Register NOR**:

- **Extended:** crnot Bx,By
- **Equivalent to:** crnor Bx,By,By

### Condition Register AND with Complement

**XL-form**

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

\[
\begin{array}{cccccc}
0 & 19 & 6 & 8 & 11 & 16 & 21 & 29 & 33 & 31 \\
\end{array}
\]

\(\text{CR}_{\text{BT+32}} \leftarrow \text{CR}_{\text{BA+32}} \& \neg \text{CR}_{\text{BB+32}}\)

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:**

\(\text{CR}_{\text{BT+32}}\)

### Condition Register OR with Complement

**XL-form**

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

\[
\begin{array}{cccccc}
0 & 19 & 6 & 8 & 11 & 16 & 21 & 31 \\
\end{array}
\]

\(\text{CR}_{\text{BT+32}} \leftarrow \text{CR}_{\text{BA+32}} \lor \neg \text{CR}_{\text{BB+32}}\)

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:**

\(\text{CR}_{\text{BT+32}}\)

### 2.5.2 Condition Register Field Instruction

**Move Condition Register Field**

**XL-form**

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

\[
\begin{array}{cccccc}
0 & 19 & 6 & 8 & 11 & 14 & 16 & 21 & 31 \\
\end{array}
\]

\(\text{CR}_{\text{BF+32:4}} \leftarrow \text{CR}_{\text{BFA+32:4}}\)

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

**Special Registers Altered:**

\(\text{CR} \text{ field BF}\)
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

<table>
<thead>
<tr>
<th>sc</th>
<th>LEV</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td>6</td>
</tr>
</tbody>
</table>

System Call Vectored

<table>
<thead>
<tr>
<th>scv</th>
<th>LEV</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td>6</td>
</tr>
</tbody>
</table>

These instructions call the system to perform a service. A complete description of these instructions can be found in Section 3.3.1 of Book III.

The first form of the instruction (\textit{sc}) provides a single system call. The second form of the instruction (\textit{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 \textit{scv} instruction modifies the Count Register, programs should treat the contents of the Count Register as undefined after executing this instruction. See Section 3.3 of Book III.

\textit{sc} serves as both a basic and an extended mnemonic. The Assembler will recognize an \textit{sc} mnemonic with one operand as the basic form, and an \textit{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 \textit{sc} should be 0.
<BHRB material moved to Chapter 8 of Book II.>
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 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>33</td>
<td>Overflow (OV)</td>
</tr>
</tbody>
</table>

- **Summary Overflow (SO)**: 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, or by other instructions (except mtspr to the XER and addex with operand CY=0) 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. addex does not alter the contents of SO.

- **Overflow (OV)**: The Overflow bit is set to indicate that an overflow has occurred during execution of an instruction. The Overflow bit can also be used as an independent Carry bit by using the addex with operand CY=0 instruction and avoiding other instructions that modify the Overflow bit (e.g., any XO-form instruction with OE=1).

XO-form Add, Subtract From, and Negate instructions having OE=1 set it to 1 if the carry out of bit M is not equal to the carry out of bit M+1, and set it to 0 otherwise.
XO-form Multiply Low and Divide instructions having OE=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.

addex with operand CY=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 mtspr to the XER) that cannot overflow.

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, mtspr 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

46:56 Reserved

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 lscbx instruction. Power ISA lacks the lscbx instruction, but some application programs that run on processors that implement Power ISA may still use lscbx, and privileged software may emulate the instruction. XER32:63 may be assigned a meaning in a future version of the architecture, when POWER compatibility for lscbx is no longer needed, so these bits should not be used for purposes other than the lscbx 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 1.11.3 on page 27.

**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**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>RT, D(RA)</th>
</tr>
</thead>
<tbody>
<tr>
<td>lbz</td>
<td></td>
</tr>
</tbody>
</table>

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

- If RA = 0 then b ← 0
- else b ← (RA)
- EA ← b + EXTS(D)
- RT ← 560 || MEM(EA, 1)
- EA is placed into register RA.
- If RA=0 or RA=RT, the instruction form is invalid.

**Special Registers Altered:**
None

### Load Byte and Zero Indexed

**X-form**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>RT, RA, RB</th>
</tr>
</thead>
<tbody>
<tr>
<td>lbzx</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RT</th>
<th>RA</th>
<th>RB</th>
<th>87</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</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)
- RT ← 560 || MEM(EA, 1)
- 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

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

<table>
<thead>
<tr>
<th>40</th>
<th>RT</th>
<th>RA</th>
<th>16</th>
<th>D</th>
</tr>
</thead>
</table>

if RA = 0 then \( b \leftarrow 0 \)
else \( b \leftarrow (RA) \)
\( \text{EA} \leftarrow b + \text{EXTS}(D) \)
\( \text{RT} \leftarrow 48_0 || \text{MEM}([\text{EA}, 2]) \)

Let the effective address (EA) be the sum (RA(0) + D). The halfword in storage addressed by EA is loaded into RT48:63. RT0:47 are set to 0.

Special Registers Altered:
None

Load Halfword and Zero Indexed

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

<table>
<thead>
<tr>
<th>31</th>
<th>RT</th>
<th>RA</th>
<th>RB</th>
<th>279</th>
</tr>
</thead>
</table>

if RA = 0 then \( b \leftarrow 0 \)
else \( b \leftarrow (RA) \)
\( \text{EA} \leftarrow b + (RB) \)
\( \text{RT} \leftarrow 48_0 || \text{MEM}([\text{EA}, 2]) \)

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.

Special Registers Altered:
None

Load Halfword and Zero with Update

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

<table>
<thead>
<tr>
<th>41</th>
<th>RT</th>
<th>RA</th>
<th>16</th>
<th>D</th>
</tr>
</thead>
</table>

\( \text{EA} \leftarrow (RA) + \text{EXTS}(D) \)
\( \text{RT} \leftarrow 48_0 || \text{MEM}([\text{EA}, 2]) \)
\( \text{RA} \leftarrow \text{EA} \)

Let the effective address (EA) be the sum (RA)+ D. The halfword in storage addressed by EA is loaded into RT48:63. RT0:47 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

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

<table>
<thead>
<tr>
<th>31</th>
<th>RT</th>
<th>RA</th>
<th>RB</th>
<th>311</th>
</tr>
</thead>
</table>

\( \text{EA} \leftarrow (RA) + (RB) \)
\( \text{RT} \leftarrow 48_0 || \text{MEM}([\text{EA}, 2]) \)
\( \text{RA} \leftarrow \text{EA} \)

Let the effective address (EA) be the sum (RA)+ (RB). The halfword in storage addressed by EA is loaded into RT48:63. RT0:47 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**

<table>
<thead>
<tr>
<th>lh</th>
<th>RT,D(RA)</th>
</tr>
</thead>
<tbody>
<tr>
<td>42</td>
<td>6</td>
</tr>
<tr>
<td>11</td>
<td>16</td>
</tr>
<tr>
<td>D</td>
<td></td>
</tr>
</tbody>
</table>

if RA = 0 then b ← 0
else b ← (RA)
EA ← b + EXTS(D)
RT ← EXTS(MEM(EA, 2))

Let the effective address (EA) be the sum (RA|0) + 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.

**Special Registers Altered:**

None

### Load Halfword Algebraic Indexed X-form

<table>
<thead>
<tr>
<th>lhax</th>
<th>RT,RA,RB</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>6</td>
</tr>
<tr>
<td>11</td>
<td>16</td>
</tr>
<tr>
<td>21</td>
<td>343</td>
</tr>
<tr>
<td>/</td>
<td></td>
</tr>
</tbody>
</table>

if RA = 0 then b ← 0
else b ← (RA)
EA ← b + (RB)
RT ← 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

### Load Halfword Algebraic with Update D-form

<table>
<thead>
<tr>
<th>lhau</th>
<th>RT,D(RA)</th>
</tr>
</thead>
<tbody>
<tr>
<td>43</td>
<td>6</td>
</tr>
<tr>
<td>11</td>
<td>16</td>
</tr>
<tr>
<td>D</td>
<td></td>
</tr>
</tbody>
</table>

EA ← (RA) + EXTS(D)
RT ← EXTS(MEM(EA, 2))
RA ← 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.

EA is placed into register RA.

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

**Special Registers Altered:**

None

### Load Halfword Algebraic with Update Indexed X-form

<table>
<thead>
<tr>
<th>lhaux</th>
<th>RT,RA,RB</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>6</td>
</tr>
<tr>
<td>11</td>
<td>16</td>
</tr>
<tr>
<td>21</td>
<td>375</td>
</tr>
<tr>
<td>/</td>
<td></td>
</tr>
</tbody>
</table>

EA ← (RA) + (RB)
RT ← EXTS(MEM(EA, 2))
RA ← 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.

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

### D-form

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

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

If \( \text{RA} = 0 \) then \( b \leftarrow 0 \)
else \( b \leftarrow (\text{RA}) \)
\( \text{EA} \leftarrow b + \text{EXTS(D)} \)
\( \text{RT} \leftarrow 32_0 | | \text{MEM(EA, 4)} \)

Let the effective address (\( \text{EA} \)) be the sum \( (\text{RA}) + D \).
The word in storage addressed by \( \text{EA} \) is loaded into \( \text{RT}_{32:63} \). \( \text{RT}_{0:31} \) are set to 0.

**Special Registers Altered:** None

### Load Word and Zero Indexed

### X-form

\[ \text{lwzx} \quad \text{RT,RA,RB} \]

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

If \( \text{RA} = 0 \) then \( b \leftarrow 0 \)
else \( b \leftarrow (\text{RA}) \)
\( \text{EA} \leftarrow b + (\text{RB}) \)
\( \text{RT} \leftarrow 32_0 | | \text{MEM(EA, 4)} \)

Let the effective address (\( \text{EA} \)) be the sum \( (\text{RA}) + (\text{RB}) \).
The word in storage addressed by \( \text{EA} \) is loaded into \( \text{RT}_{32:63} \). \( \text{RT}_{0:31} \) are set to 0.

**Special Registers Altered:** None

## Load Word and Zero with Update

### D-form

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

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

\( \text{EA} \leftarrow (\text{RA}) + \text{EXTS(D)} \)
\( \text{RT} \leftarrow 32_0 | | \text{MEM(EA, 4)} \)
\( \text{RA} \leftarrow \text{EA} \)

Let the effective address (\( \text{EA} \)) be the sum \( (\text{RA}) + D \).
The word in storage addressed by \( \text{EA} \) is loaded into \( \text{RT}_{32:63} \). \( \text{RT}_{0:31} \) are set to 0.

\( \text{EA} \) is placed into register \( \text{RA} \).

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

**Special Registers Altered:** None

### Load Word and Zero with Update Indexed

### X-form

\[ \text{lwzux} \quad \text{RT,RA,RB} \]

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

\( \text{EA} \leftarrow (\text{RA}) + (\text{RB}) \)
\( \text{RT} \leftarrow 32_0 | | \text{MEM(EA, 4)} \)
\( \text{RA} \leftarrow \text{EA} \)

Let the effective address (\( \text{EA} \)) be the sum \( (\text{RA}) + (\text{RB}) \).
The word in storage addressed by \( \text{EA} \) is loaded into \( \text{RT}_{32:63} \). \( \text{RT}_{0:31} \) are set to 0.

\( \text{EA} \) is placed into register \( \text{RA} \).

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

**Special Registers Altered:** None
3.3.2.1 64-bit Fixed-Point Load Instructions

**Load Word Algebraic**

\[ \text{lwa } \text{RT, DS(RA)} \]

<table>
<thead>
<tr>
<th>58</th>
<th>6</th>
<th>32</th>
<th>2</th>
</tr>
</thead>
</table>

if RA = 0 then b ← 0
else b ← (RA)

EA ← b + EXTS(DS || 0b00)
RT ← EXTS(MEM(EA, 4))

Let the effective address (EA) be the sum \((RA|0)+ (DS||0b00)\). 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**

\[ \text{l wax } \text{RT, RA, RB} \]

<table>
<thead>
<tr>
<th>31</th>
<th>6</th>
<th>16</th>
<th>21</th>
<th>341</th>
<th>/</th>
</tr>
</thead>
</table>

if RA = 0 then b ← 0
else b ← (RA)

EA ← b + (RB)
RT ← 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**

\[ \text{l waxu } \text{RT, RA, RB} \]

<table>
<thead>
<tr>
<th>31</th>
<th>6</th>
<th>16</th>
<th>21</th>
<th>373</th>
<th>/</th>
</tr>
</thead>
</table>

EA ← (RA) + (RB)
RT ← EXTS(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 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
Load Doubleword  

**DS-form**

\[
\text{ld \ RT,DS(RA)} \\
\]

<table>
<thead>
<tr>
<th>58</th>
<th>RT</th>
<th>RA</th>
<th>DS</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>11</td>
<td>16</td>
<td>30 31</td>
</tr>
</tbody>
</table>

if RA = 0 then b ← 0 
else b ← (RA) 
EA ← b + EXTS(DS || 0b00) 
RT ← MEM(EA, 8)

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

**Special Registers Altered:** 
None

Load Doubleword Indexed  

**X-form**

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

<table>
<thead>
<tr>
<th>31</th>
<th>RT</th>
<th>RA</th>
<th>RB</th>
<th>21</th>
<th>/</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>31</td>
<td>8</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) 
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.

**Special Registers Altered:** 
None

Load Doubleword with Update  

**DS-form**

\[
\text{ldu \ RT,DS(RA)} \\
\]

<table>
<thead>
<tr>
<th>58</th>
<th>RT</th>
<th>RA</th>
<th>DS</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>11</td>
<td>16</td>
<td>1</td>
</tr>
</tbody>
</table>

EA ← (RA) + EXTS(DS || 0b00) 
RT ← MEM(EA, 8) 
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**

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

<table>
<thead>
<tr>
<th>31</th>
<th>RT</th>
<th>RA</th>
<th>RB</th>
<th>53</th>
<th>/</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>31</td>
<td>8</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

EA ← (RA) + (RB) 
RT ← 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 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.

#### Store Byte

```
store RS, D(RA)
```

<table>
<thead>
<tr>
<th>D-form</th>
</tr>
</thead>
<tbody>
<tr>
<td>38</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

if RA = 0 then b ← 0
else b ← (RA)
EA ← b + EXTS(D)
MEM(EA, 1) ← (RS)56:63

Let the effective address (EA) be the sum (RA)+ D. (RS)56:63 are stored into the byte in storage addressed by EA.

**Special Registers Altered:**
None

#### Store Byte Indexed

```
store X RS, RA, RB
```

<table>
<thead>
<tr>
<th>X-form</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
</tr>
<tr>
<td></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)+ (RB). (RS)56:63 are stored into the byte in storage addressed by EA.

**Special Registers Altered:**
None

#### Store Byte with Update

```
store u RS, D(RA)
```

<table>
<thead>
<tr>
<th>D-form</th>
</tr>
</thead>
<tbody>
<tr>
<td>39</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

EA ← (RA) + EXTS(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

```
store u X RS, RA, RB
```

<table>
<thead>
<tr>
<th>X-form</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
</tr>
<tr>
<td></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**

<table>
<thead>
<tr>
<th></th>
<th>D-form</th>
<th>X-form</th>
</tr>
</thead>
<tbody>
<tr>
<td>sth</td>
<td>RS,D(RA)</td>
<td>sthx RS,RA,RB</td>
</tr>
<tr>
<td>44</td>
<td>6</td>
<td>31</td>
</tr>
<tr>
<td>6</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>16</td>
<td>D</td>
<td>21</td>
</tr>
</tbody>
</table>
| 31 |                 | 31             | (RS)48:63 are stored into the halfword in storage addressed by EA.

Let the effective address (EA) be the sum (RA(0)+ D. (RS)48:63 are stored into the halfword in storage addressed by EA.

Special Registers Altered:
None

*Store Halfword Indexed*  

<table>
<thead>
<tr>
<th></th>
<th>D-form</th>
<th>X-form</th>
</tr>
</thead>
<tbody>
<tr>
<td>sthx</td>
<td>RS,RA,RB</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>6</td>
<td>31</td>
</tr>
<tr>
<td>6</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>16</td>
<td>RA</td>
<td>21</td>
</tr>
</tbody>
</table>
| 407|                 | 31             | (RS)48:63 are stored into the halfword in storage addressed by EA.

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*  

<table>
<thead>
<tr>
<th></th>
<th>D-form</th>
</tr>
</thead>
<tbody>
<tr>
<td>sthu</td>
<td>RS,D(RA)</td>
</tr>
<tr>
<td>45</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>11</td>
</tr>
<tr>
<td>16</td>
<td>D</td>
</tr>
</tbody>
</table>
| 31 |                 | (RS)48:63 are stored into the halfword in storage addressed by 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*  

<table>
<thead>
<tr>
<th></th>
<th>X-form</th>
</tr>
</thead>
<tbody>
<tr>
<td>sthux</td>
<td>RS,RA,RB</td>
</tr>
<tr>
<td>31</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>11</td>
</tr>
<tr>
<td>16</td>
<td>RA</td>
</tr>
<tr>
<td>21</td>
<td>RB</td>
</tr>
</tbody>
</table>
| 439|                 | (RS)48:63 are stored into the halfword in storage addressed by 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  

\[
\text{stw} \quad RS, D(RA)
\]

<table>
<thead>
<tr>
<th></th>
<th>36</th>
<th>RS</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>
<td>31</td>
</tr>
</tbody>
</table>

if RA = 0 then b ← 0
else b ← (RA)
EA ← b + EXTS(D)
MEM(EA, 4) ← (RS)_{32:63}

Let the effective address (EA) be the sum (RA|0)+ D. (RS)_{32:63} are stored into the word in storage addressed by EA.

Special Registers Altered:
None

Store Word Indexed  

\[
\text{stwx} \quad RS, RA, RB
\]

|   | 31 | RS | RA | RB | 151 | /
<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>31</td>
</tr>
</tbody>
</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|0)+ (RB). (RS)_{32:63} are stored into the word in storage addressed by EA.

Special Registers Altered:
None

Store Word with Update  

\[
\text{stwu} \quad RS, D(RA)
\]

<table>
<thead>
<tr>
<th></th>
<th>37</th>
<th>RS</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>
<td>31</td>
</tr>
</tbody>
</table>

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  

\[
\text{stwux} \quad RS, RA, RB
\]

|   | 31 | RS | RA | RB | 183 | /
<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>31</td>
</tr>
</tbody>
</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**

<table>
<thead>
<tr>
<th>std</th>
<th>RS,DS(RA)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>62</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>0</th>
<th>30</th>
<th>31</th>
</tr>
</thead>
</table>

if RA = 0 then b ← 0
else b ← (RA)
EA ← b + EXTS(DS || 0b00)
MEM(EA, 8) ← (RS)

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

**Special Registers Altered:**

None

**Store Doubleword Indexed**

<table>
<thead>
<tr>
<th>stdx</th>
<th>RS,RA,RB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>31</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>30</th>
<th>31</th>
</tr>
</thead>
</table>

if RA = 0 then b ← 0
else b ← (RA)
EA ← b + (RB)
MEM(EA, 8) ← (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**

<table>
<thead>
<tr>
<th>stdu</th>
<th>RS,DS(RA)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>62</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>1</th>
<th>30</th>
<th>31</th>
</tr>
</thead>
</table>

EA ← (RA) + EXTS(DS || 0b00)
MEM(EA, 8) ← (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**

<table>
<thead>
<tr>
<th>stdux</th>
<th>RS,RA,RB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>31</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>181</th>
<th>31</th>
</tr>
</thead>
</table>

EA ← (RA) + (RB)
MEM(EA, 8) ← (RS)
RA ← 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.

In the preferred form of the 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.

### Load Quadword

<table>
<thead>
<tr>
<th>DQ-form</th>
</tr>
</thead>
<tbody>
<tr>
<td>$lq$</td>
</tr>
<tr>
<td>$RTp, DQ(RA)$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>56</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>11</td>
</tr>
<tr>
<td>16</td>
<td></td>
</tr>
</tbody>
</table>

if $RA = 0$ then $b \leftarrow 0$
else $b \leftarrow [RA]$

$EA \leftarrow b + EXTS(DQ || 0b0000)$
$RTp \leftarrow MEM(EA, 16)$

Let the effective address (EA) be the sum $(RA)[0]+(DQ)[0b0000]$. The quadword in storage addressed by EA is loaded into register pair 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.

---

### 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 Load Doubleword or 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

<table>
<thead>
<tr>
<th>DQ-form</th>
</tr>
</thead>
<tbody>
<tr>
<td>$lq$</td>
</tr>
<tr>
<td>$RTp, DQ(RA)$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>56</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>11</td>
</tr>
<tr>
<td>16</td>
<td></td>
</tr>
</tbody>
</table>

if $RA = 0$ then $b \leftarrow 0$
else $b \leftarrow [RA]$

$EA \leftarrow b + EXTS(DQ || 0b0000)$
$RTp \leftarrow MEM(EA, 16)$

Let the effective address (EA) be the sum $(RA)[0]+(DQ)[0b0000]$. The quadword in storage addressed by EA is loaded into register pair 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.

---

### Programming Note

In versions of the architecture prior to V. 2.07, this instruction was privileged.

### Special Registers Altered:

None
**Store Quadword**

\[
\text{stq} \quad \text{RSp,DS(RA)}
\]

<table>
<thead>
<tr>
<th></th>
<th></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>30 31</td>
</tr>
</tbody>
</table>

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

\(EA \leftarrow b + \text{EXTS}(DS || 0b00)\)

\(\text{MEM}(EA, 16) \leftarrow \text{RSp}\)

Let the effective address (EA) be the sum (RA[0]+(DS||0b00). The contents of register pair RSp are 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.

**Programming Note**

In versions of the architecture prior to V. 2.07, this instruction was privileged.

**Special Registers Altered:**

None
3.3.5 Fixed-Point Load and Store with Byte Reversal Instructions

**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.

---

**Load Halfword Byte-Reverse Indexed X-form**

\[
\text{lhbrx} \quad RT, RA, RB
\]

- If RA = 0 then b ← 0
- Else b ← (RA)
- \( EA \leftarrow b + (RB) \)
- load_data ← MEM(EA, 2)
- \( RT \leftarrow 480 \ || \text{load_data}_{8:15} \ || \text{load_data}_{0:7} \)

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 RT\(_{56:63}\). Bits 8:15 of the halfword in storage addressed by EA are loaded into RT\(_{48:55}\). RT\(_{0:47}\) are set to 0.

**Special Registers Altered:**
None

**Store Halfword Byte-Reverse Indexed X-form**

\[
\text{sthbrx} \quad RS, RA, RB
\]

- If RA = 0 then b ← 0
- Else b ← (RA)
- \( EA \leftarrow b + (RB) \)
- 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

---

**Load Word Byte-Reverse Indexed X-form**

\[
\text{lwbrx} \quad RT, RA, RB
\]

- If RA = 0 then b ← 0
- Else b ← (RA)
- \( EA \leftarrow b + (RB) \)
- load_data ← MEM(EA, 4)
- \( RT \leftarrow 320 \ || \text{load_data}_{24:31} \ || \text{load_data}_{16:23} \ || \text{load_data}_{8:15} \ || \text{load_data}_{0:7} \)

Let the effective address (EA) be the sum (RA|0)+(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\(_{32:39}\). Bits 24:31 of the word in storage addressed by EA are loaded into RT\(_{32:39}\). RT\(_{0:31}\) are set to 0.

**Special Registers Altered:**
None

**Store Word Byte-Reverse Indexed X-form**

\[
\text{stwbrx} \quad RS, RA, RB
\]

- If RA = 0 then b ← 0
- Else b ← (RA)
- \( EA \leftarrow b + (RB) \)
- MEM(EA, 4) ← (RS)\(_{56:63}\) \ || (RS)\(_{48:55}\) \ || (RS)\(_{40:47}\) \ || (RS)\(_{32:39}\)

Let the effective address (EA) be the sum (RA|0)+(RB). (RS)\(_{56:63}\) are stored into bits 0:7 of the word in storage addressed by EA. (RS)\(_{48:55}\) are stored into bits 8:15 of the word in storage addressed by EA. (RS)\(_{40:47}\) are stored into bits 16:23 of the word in storage addressed by EA. (RS)\(_{32:39}\) are stored into bits 24:31 of the word in storage addressed by EA.

**Special Registers Altered:**
None

---

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.

In some implementations, the Load Byte-Reverse instructions may have greater latency than other Load instructions.
3.3.5.1 64-Bit Load and Store with Byte Reversal Instructions

**Load Doubleword Byte-Reverse Indexed**

**X-form**

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

if RA = 0 then b ← 0
else b ← (RA)
EA ← b + (RB)
load_data ← MEM(EA, 8)

\[
\begin{align*}
\text{RT} &\leftarrow \text{load_data}_{56:63} \\
&\quad | \text{load_data}_{48:55} \\
&\quad | \text{load_data}_{40:47} \\
&\quad | \text{load_data}_{32:39} \\
&\quad | \text{load_data}_{24:31} \\
&\quad | \text{load_data}_{16:23} \\
&\quad | \text{load_data}_{8:15} \\
&\quad | \text{load_data}_{0:7}
\end{align*}
\]

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**

\[
\text{stdbrx } RS, RA, RB
\]

if RA = 0 then b ← 0
else b ← (RA)
EA ← b + (RB)
MEM(EA, 8) ← (RS)56:63 | (RS)48:55

\[
\begin{align*}
\text{RS} &\leftarrow (RS)_{40:47} \\
&\quad | (RS)_{32:39} \\
&\quad | (RS)_{24:31} \\
&\quad | (RS)_{16:23} \\
&\quad | (RS)_{8:15} \\
&\quad | (RS)_{0:7}
\end{align*}
\]

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  

<table>
<thead>
<tr>
<th>D-form</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>lmw</strong></td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

if RA = 0 then b ← 0  
else b ← (RA)  
EA ← b + EXTS(D)  
r ← RT  
do while r ≤ 31  
   GPR(r) ← \(32_0\) || 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  

<table>
<thead>
<tr>
<th>D-form</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>stmw</strong></td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

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 1.9.1, “Preferred Instruction Forms” on page 23. In the preferred forms, register usage satisfies the following rules.

- RS = 4 or 5
- RT = 4 or 5
- last register loaded/stored ≤ 12

For some implementations, using GPR 4 for RS and RT may result in slightly faster execution than using GPR 5.
### Load String Word Immediate

**X-form**

<table>
<thead>
<tr>
<th>lswi</th>
<th>RT, RA, NB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>31</td>
</tr>
<tr>
<td>5</td>
<td>11</td>
</tr>
<tr>
<td>16</td>
<td>21</td>
</tr>
<tr>
<td>597</td>
<td>/</td>
</tr>
</tbody>
</table>

if RA = 0 then EA $\leftarrow 0$
else
  EA $\leftarrow (RA)$
if NB = 0 then $n \leftarrow 32$
else
  $n \leftarrow NB$
$r \leftarrow RT - 1$
i $\leftarrow 32$
do while $n > 0$
  if $i = 32$ then
    $r \leftarrow r + 1 \pmod{32}$
    GPR$(r)$ $\leftarrow 0$
    GPR$(r)[i:i+7] \leftarrow \text{MEM}(EA, 1)$
i $\leftarrow i + 8$
  if $i = 64$ then $i \leftarrow 32$
i $\leftarrow n - 1$

Let the effective address (EA) be (RA|0). Let $n = NB$ if $NB \neq 0$, $n = 32$ if $NB = 0$; $n$ is the number of bytes to load. Let $nr = \lceil n / 4 \rceil$; $nr$ 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**

<table>
<thead>
<tr>
<th>lswx</th>
<th>RT, RA, RB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>31</td>
</tr>
<tr>
<td>5</td>
<td>11</td>
</tr>
<tr>
<td>16</td>
<td>21</td>
</tr>
<tr>
<td>533</td>
<td>/</td>
</tr>
</tbody>
</table>

if RA = 0 then $b \leftarrow 0$
else
  $b \leftarrow (RA)$
EA $\leftarrow b + (RB)$
n $\leftarrow \text{XER}_{57:63}$
i $\leftarrow RT - 1$
i $\leftarrow 32$
RT $\leftarrow $ undefined
do while $n > 0$
  if $i = 32$ then
    $r \leftarrow r + 1 \pmod{32}$
    GPR$(r)$ $\leftarrow 0$
    GPR$(r)[i:i+7] \leftarrow \text{MEM}(EA, 1)$
i $\leftarrow i + 8$
  if $i = 64$ then $i \leftarrow 32$
i $\leftarrow n - 1$

Let the effective address (EA) be the sum (RA|0)+ (RB). Let $n = \text{XER}_{57:63}$; $n$ is the number of bytes to load. Let $nr = \lceil n / 4 \rceil$; $nr$ 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
| 31 | 26 | 21 | 16 | 11 | 6 | 0 |
---|---|---|---|---|---|---|
  0 | RS | RA | NB | | | |
```

if RA = 0 then EA ← 0
else EA ← (RA)
if NB = 0 then n ← 32
else n ← NB
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

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**

```
stwx   RS,RA,RB
| 31 | 26 | 21 | 16 | 11 | 6 | 0 |
---|---|---|---|---|---|---|
  0 | RS | RA | RB | | | |
```

if RA = 0 then b ← 0
else b ← (RA)
EA ← b + (RB)
n ← XER 57:63
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

Let the effective address (EA) be the sum (RA|0)+ (RB). Let n = XER 57:63; 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 `addic`, 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, divide, 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

<table>
<thead>
<tr>
<th>Add Immediate</th>
<th>D-form</th>
</tr>
</thead>
<tbody>
<tr>
<td>addi</td>
<td>RT,RA,SI</td>
</tr>
</tbody>
</table>

#### D-form

<table>
<thead>
<tr>
<th></th>
<th>RT</th>
<th>RA</th>
<th>SI</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>8</td>
<td>11</td>
<td>16</td>
</tr>
</tbody>
</table>

if RA = 0 then RT ← EXTS(SI)
else RT ← (RA) + EXTS(SI)

The sum (RA(0) + SI) is placed into register RT.

**Special Registers Altered:**

None

**Extended Mnemonics:**

Examples of extended mnemonics for `Add Immediate`:

```
  Extended:   Equivalent to:
  li    Rx,value   addi    Rx,0,value
  ia    Rx,disp(Ry)  addi   Rx,Ry,disp
  subi   Rx,Ry,value  addi   Rx,Ry,-value
```

**Programming Note**

`addi, addis, add, and subf` are the preferred instructions for addition and subtraction, because they set few status bits.

Notice that `addi` and `addis` use the value 0, not the contents of GPR 0, if RA=0.

### Add Immediate Shifted

<table>
<thead>
<tr>
<th>Add Immediate Shifted</th>
<th>D-form</th>
</tr>
</thead>
<tbody>
<tr>
<td>addis</td>
<td>RT,RA,SI</td>
</tr>
</tbody>
</table>

#### D-form

<table>
<thead>
<tr>
<th></th>
<th>RT</th>
<th>RA</th>
<th>SI</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>8</td>
<td>11</td>
<td>16</td>
</tr>
</tbody>
</table>

if RA = 0 then RT ← EXTS(SI || 160)
else RT ← (RA) + EXTS(SI || 160)

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:   Equivalent to:
  lis    Rx,value  addis    Rx,0,value
  subis   Rx,Ry,value  addis   Rx,Ry,-value
```
**Add PC Immediate Shifted**

**DX-form**

```
addpcis  RT,D
```

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>26</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>19</td>
<td>RT</td>
<td>d1</td>
<td>d0</td>
<td>2</td>
<td>d2</td>
<td></td>
</tr>
</tbody>
</table>
```

D ← d0 || d1 || d2
RT ← NIA + EXTS(D || 160)

The sum of NIA + (D || 0x0000) is placed into register RT.

**Special Registers Altered:**
None

**Extended Mnemonics:**

Examples of extended mnemonics for Add PC Immediate Shifted:

<table>
<thead>
<tr>
<th>Extended:</th>
<th>Equivalent to:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inia Rx</td>
<td>addpcis Rx,0</td>
</tr>
<tr>
<td>subpcis Rx,value</td>
<td>addpcis Rx,-value</td>
</tr>
</tbody>
</table>
Add

<table>
<thead>
<tr>
<th>Add</th>
<th>XO-form</th>
<th>Subtract From</th>
<th>XO-form</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>RT,RA,RB (OE=0 Rc=0)</td>
<td>subf</td>
<td>RT,RA,RB (OE=0 Rc=0)</td>
</tr>
<tr>
<td>add.</td>
<td>RT,RA,RB (OE=0 Rc=1)</td>
<td>sub.</td>
<td>RT,RA,RB (OE=0 Rc=1)</td>
</tr>
<tr>
<td>addo</td>
<td>RT,RA,RB (OE=1 Rc=0)</td>
<td>subfo</td>
<td>RT,RA,RB (OE=1 Rc=0)</td>
</tr>
<tr>
<td>addo.</td>
<td>RT,RA,RB (OE=1 Rc=1)</td>
<td>subfo.</td>
<td>RT,RA,RB (OE=1 Rc=1)</td>
</tr>
</tbody>
</table>

The sum (RA) + (RB) 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: Equivalent to:
- sub Rx,Ry,Rz subf Rx,Rz,Ry

Add Immediate Carrying

<table>
<thead>
<tr>
<th>Add Immediate Carrying</th>
<th>D-form</th>
</tr>
</thead>
<tbody>
<tr>
<td>addic</td>
<td>RT,RA,SI</td>
</tr>
</tbody>
</table>

The sum (RA) + SI is placed into register RT.

Special Registers Altered:
- CA CA32

Extended Mnemonics:
Example of extended mnemonics for Add Immediate Carrying:

Extended: Equivalent to:
- subic Rx,Ry,value addic Rx,Ry,−value

Add Immediate Carrying and Record

<table>
<thead>
<tr>
<th>Add Immediate Carrying and Record</th>
<th>D-form</th>
</tr>
</thead>
<tbody>
<tr>
<td>addic.</td>
<td>RT,RA,SI</td>
</tr>
</tbody>
</table>

The sum (RA) + SI is placed into register RT.

Special Registers Altered:
- CR0 CA CA32

Extended Mnemonics:
Example of extended mnemonics for Add Immediate Carrying and Record:

Extended: Equivalent to:
- subic Rx,Ry,value addic Rx,Ry,−value
**Subtract From Immediate Carrying**

*D-form*

\[ \text{subfic} \quad RT, RA, SI \]

\[ \begin{array}{c|ccc|c}
8 & RT & RA & SI \\
\hline
0 & 6 & 11 & 16 & 31
\end{array} \]

\( RT \leftarrow \neg (RA) + \text{EXTS(SI)} + 1 \)

The sum \( \neg (RA) + SI + 1 \) is placed into register RT.

**Special Registers Altered:**

CA, CA32

---

**Add Carrying**

* XO-form *

\[ \text{addc} \quad RT, RA, RB \]  \( \text{(OE=0 Rc=0)} \)

\[ \text{addc.} \quad RT, RA, RB \]  \( \text{(OE=0 Rc=1)} \)

\[ \text{addco} \quad RT, RA, RB \]  \( \text{(OE=1 Rc=0)} \)

\[ \text{addco.} \quad RT, RA, RB \]  \( \text{(OE=1 Rc=1)} \)

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

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 *

\[ \text{subfc} \quad RT, RA, RB \]  \( \text{(OE=0 Rc=0)} \)

\[ \text{subfc.} \quad RT, RA, RB \]  \( \text{(OE=0 Rc=1)} \)

\[ \text{subfco} \quad RT, RA, RB \]  \( \text{(OE=1 Rc=0)} \)

\[ \text{subfco.} \quad RT, RA, RB \]  \( \text{(OE=1 Rc=1)} \)

\[ RT \leftarrow \neg (RA) + (RB) + 1 \]

The sum \( \neg (RA) + (RB) + 1 \) 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*:

Extended:  \( \text{subc} \quad Rx, Ry, Rz \)

Equivalent to:  \( \text{subfc} \quad Rx, Rz, Ry \)
### Add Extended XO-form

<table>
<thead>
<tr>
<th></th>
<th>RT,RA,RB</th>
<th>RT,RA,RB</th>
<th>RT,RA,RB</th>
<th>RT,RA,RB</th>
</tr>
</thead>
<tbody>
<tr>
<td>OE</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Rc</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th></th>
<th>RT,RA,RB</th>
<th>RT,RA,RB</th>
<th>RT,RA,RB</th>
<th>RT,RA,RB</th>
</tr>
</thead>
<tbody>
<tr>
<td>OE</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Rc</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

The sum \(\neg(RA) + (RB) + CA\) 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

<table>
<thead>
<tr>
<th></th>
<th>RT,RA</th>
<th>RT,RA</th>
<th>RT,RA</th>
<th>RT,RA</th>
</tr>
</thead>
<tbody>
<tr>
<td>OE</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Rc</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

The sum \((RA) + CA - 1\) 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

<table>
<thead>
<tr>
<th></th>
<th>RT,RA</th>
<th>RT,RA</th>
<th>RT,RA</th>
<th>RT,RA</th>
</tr>
</thead>
<tbody>
<tr>
<td>OE</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Rc</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

The sum \(\neg(RA) + CA - 1\) is placed into register RT.

**Special Registers Altered:**
- CA
- CA32
- CR0 (if Rc=1)
- SO
- OV
- OV32 (if OE=1)
Add Extended using alternate carry bit
Z23-form

\[
\text{addex } RT, RA, RB, CY
\]

\[
\begin{array}{|c|c|c|c|c|}
\hline
& RT & RA & RB & CY \\
\hline
31 & 6 & 11 & 16 & 21 \\
170 & & & & 31 \\
\hline
\end{array}
\]

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 subfzoe, subtracting any operand from itself.

Add to Zero Extended

\[
\begin{align*}
\text{addze } & \text{ RT, RA } \\
\text{addzee } & \text{ RT, RA } \\
\text{addzeo } & \text{ RT, RA } \\
\text{addzeoo } & \text{ RT, RA }
\end{align*}
\]

\[
\begin{array}{|c|c|c|c|c|c|}
\hline
& RT & RA & /// & OE & 202 & RC \\
\hline
31 & 6 & 11 & 16 & 21 & 22 & 31 \\
\hline
\end{array}
\]

RT ← (RA) + CA

The sum (RA) + CA is placed into register RT.

Special Registers Altered:

- CA
- CA32
- CR0
- SO
- OV
- OV32

(if CY = 0)

Subtract From Zero Extended

\[
\begin{align*}
\text{subfze } & \text{ RT, RA } \\
\text{subfzee } & \text{ RT, RA } \\
\text{subfzeo } & \text{ RT, RA } \\
\text{subfzeoo } & \text{ RT, RA }
\end{align*}
\]

\[
\begin{array}{|c|c|c|c|c|c|}
\hline
& RT & RA & /// & OE & 200 & RC \\
\hline
31 & 6 & 11 & 16 & 21 & 22 & 31 \\
\hline
\end{array}
\]

RT ← ¬(RA) + CA

The sum ¬(RA) + CA is placed into register RT.

Special Registers Altered:

- CA
- CA32
- CR0
- SO
- OV
- OV32

(if CY = 0)

Negate

\[
\begin{align*}
\text{neg } & \text{ RT, RA } \\
\text{neg } & \text{ RT, RA } \\
\text{nego } & \text{ RT, RA } \\
\text{nego } & \text{ RT, RA }
\end{align*}
\]

\[
\begin{array}{|c|c|c|c|c|c|}
\hline
& RT & RA & /// & OE & 104 & RC \\
\hline
31 & 6 & 11 & 16 & 21 & 22 & 31 \\
\hline
\end{array}
\]

RT ← ¬(RA) + 1

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 and OV32 are 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
- SO
- OV
- OV32

(if CY = 0)


### Multiply Low Immediate

**D-form**

\[ \text{mulli } RT, RA, SI \]

| \( \text{prod}_{0:127} \) | \( \leftrightarrow \) | \( (RA) \times \text{EXTS}(SI) \) |
|-------------------------|----------------------|
| \( RT \) | \( \leftarrow \) | \( \text{prod}_{64:127} \) |

The 64-bit first operand is \( (RA) \). The 64-bit second operand is the sign-extended value of the SI field. The low-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:** None

### Multiply Low Word

**XO-form**

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

<table>
<thead>
<tr>
<th>( \text{OE} ) = 0 (Rc=0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{mullw. } RT, RA, RB )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( \text{OE} ) = 1 (Rc=0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{mullwo } RT, RA, RB )</td>
</tr>
</tbody>
</table>

\[ RT_{32:63} \leftarrow (RA)_{32:63} \times (RB)_{32:63} \]

The 32-bit operands are the low-order 32 bits of \( RA \) and of \( RB \). The 64-bit product of the operands is placed into register \( RT_{32:63} \). The contents of \( RT_{0:31} \) 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 Rc=1)

### Multiply High Word

**XO-form**

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

<table>
<thead>
<tr>
<th>(Rc=0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{mulhw. } RT, RA, RB )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(Rc=1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{mulhd } RT, RA, RB )</td>
</tr>
</tbody>
</table>

\[ RT_{32:63} \leftarrow (RA)_{32:63} \times (RB)_{32:63} \]

The 32-bit operands are the low-order 32 bits of \( RA \) and of \( RB \). The high-order 32 bits of the 64-bit product of the operands are placed into \( RT_{32:63} \). The contents of \( RT_{0:31} \) 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 Rc=1)

### Multiply High Word Unsigned

**XO-form**

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

<table>
<thead>
<tr>
<th>(Rc=0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{mulhwu. } RT, RA, RB )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(Rc=1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{mulhwd } RT, RA, RB )</td>
</tr>
</tbody>
</table>

\[ RT_{32:63} \leftarrow (RA)_{32:63} \times (RB)_{32:63} \]

The 32-bit operands are the low-order 32 bits of \( RA \) and of \( RB \). The high-order 32 bits of the 64-bit product of the operands are placed into \( RT_{32:63} \). The contents of \( RT_{0:31} \) are undefined.

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 (bits 0:2 undefined in 64-bit mode) (if Rc=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.
**Divide Word**

<table>
<thead>
<tr>
<th>Command</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>divw</code></td>
<td>RT, RA, RB</td>
</tr>
<tr>
<td><code>divw</code></td>
<td>RT, RA, RB</td>
</tr>
<tr>
<td><code>divw</code></td>
<td>RT, RA, RB</td>
</tr>
<tr>
<td><code>divw</code></td>
<td>RT, RA, RB</td>
</tr>
</tbody>
</table>

**XO-form**

- `divw` (OE=0 Rc=0)
- `divw` (OE=0 Rc=1)
- `divw` (OE=1 Rc=0)
- `divw` (OE=1 Rc=1)

**Dividend**

- `(RA)32:63`

**Divisor**

- `(RB)32:63`

**Quotient**

- `RT32:63`

- `RT0:31` 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 `RT0:31` 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 \(0 \leq r < |\text{divisor}|\) 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

- \(<\text{anything}> \times 0\)
- \(<\text{anything}> \div 0\)

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`.

- `divw` RT, RA, RB # RT = quotient
- `mullw` RT, RT, RB # RT = quotient x divisor
- `subf` RT, RT, RA # RT = remainder

---

**Divide Word Unsigned**

<table>
<thead>
<tr>
<th>Command</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>divwu</code></td>
<td>RT, RA, RB</td>
</tr>
<tr>
<td><code>divwu</code></td>
<td>RT, RA, RB</td>
</tr>
<tr>
<td><code>divwu</code></td>
<td>RT, RA, RB</td>
</tr>
<tr>
<td><code>divwu</code></td>
<td>RT, RA, RB</td>
</tr>
</tbody>
</table>

**XO-form**

- `divwu` (OE=0 Rc=0)
- `divwu` (OE=0 Rc=1)
- `divwu` (OE=1 Rc=0)
- `divwu` (OE=1 Rc=1)

**Dividend**

- `(RA)32:63`

**Divisor**

- `(RB)32:63`

**Quotient**

- `RT32:63`

- `RT0:31` 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 `RT0:31` are undefined. 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 an attempt is made to perform the division

- `<\text{anything}> \div 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 this case, 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 unsigned remainder of dividing `(RA)32:63` by `(RB)32:63` can be computed as follows.

- `divwu` RT, RA, RB # RT = quotient
- `mullw` RT, RT, RB # RT = quotient x divisor
- `subf` RT, RT, RA # RT = remainder
Divide Word Extended XO-form

\[
divwe \quad RT, RA, RB \quad (OE=0 \quad Rc=0)
divwe. RT, RA, RB \quad (OE=0 \quad Rc=1)
divweo RT, RA, RB \quad (OE=1 \quad Rc=0)
divweo. RT, RA, RB \quad (OE=1 \quad Rc=1)
\]

The 64-bit dividend is (RA)_{32:63} || 320. The 32-bit divisor is (RB)_{32:63}. If the quotient can be represented in 32 bits, it 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

\[
dividend = (quotient \times divisor) + r
\]

where \(0 \leq r < |divisor|\) if the dividend is nonnegative, and \(-|divisor| < r \leq 0\) if the dividend is negative.

If the quotient cannot be represented in 32 bits, or if an attempt is made to perform the division

<anything> \div 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:
C0 (bits 0:2 undefined in 64-bit mode) (if Rc=1)
SO OV OV32 (if OE=1)

Divide Word Extended Unsigned XO-form

\[
divweu \quad RT, RA, RB \quad (OE=0 \quad Rc=0)
divweu. RT, RA, RB \quad (OE=0 \quad Rc=1)
divweuo RT, RA, RB \quad (OE=1 \quad Rc=0)
divweuo. RT, RA, RB \quad (OE=1 \quad Rc=1)
\]

The 64-bit dividend is (RA)_{32:63} || 320. The 32-bit divisor is (RB)_{32:63}. If the quotient can be represented in 32 bits, it 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 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

\[
dividend = (quotient \times divisor) + r
\]

where \(0 \leq r < divisor\).

If (RA) \geq (RB), or if an attempt is made to perform the division

<anything> \div 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:
C0 (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. \( q_1 \leftarrow \text{divweu} \, \text{Dh, Dv} \)  
4. \( r_1 \leftarrow -(q_1 \times Dv) \)  
   \# remainder of step 1 divide operation (see Note 1)
5. \( q_2 \leftarrow \text{divwu} \, \text{Dl, Dv} \)
6. \( r_2 \leftarrow \text{Dl} - (q_2 \times Dv) \)  
   \# remainder of step 2 divide operation
7. \( Q \leftarrow q_1 + q_2 \)
8. \( R \leftarrow r_1 + r_2 \)
9. if \( (R < r_2) \) \| \( (R \geq Dv) \) 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  # 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 ?
blt *+12  # must adjust Q and R if yes
cmp lw r4,r6  # R \geq Dv ?
blt *+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 || 320 - (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 || 320 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

<table>
<thead>
<tr>
<th>31</th>
<th>RT</th>
<th>RA</th>
<th>RB</th>
<th>779</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>/</td>
</tr>
</tbody>
</table>

- dividend_{32:31} ← (RA)_{32:31}
- divisor_{32:31} ← (RB)_{32:31}
- RT_{32:31} ← dividend \% divisor
- RT_{0:31} ← undefined

The 32-bit dividend is (RA)_{32:31}. The 32-bit divisor is (RB)_{32:31}. The 32-bit quotient is placed into RT_{32:31}. 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

\[0 \times 8000,0000 \% -1\]
\[<\text{anything}> \% 0\]

then the contents of register RT are undefined.

**Special Registers Altered:**

None

**Modulo Unsigned Word X-form**

moduw RT,RA,RB

<table>
<thead>
<tr>
<th>31</th>
<th>RT</th>
<th>RA</th>
<th>RB</th>
<th>267</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>/</td>
</tr>
</tbody>
</table>

- dividend_{32:31} ← (RA)_{32:31}
- divisor_{32:31} ← (RB)_{32:31}
- RT_{32:31} ← dividend \% divisor
- RT_{0:31} ← undefined

The 32-bit dividend is (RA)_{32:31}. The 32-bit divisor is (RB)_{32:31}. The 32-bit quotient is placed into RT_{32:31}. 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

darn RT,L

RT \leftarrow \text{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 0:0xFFFFFFFF_FFFFFFFE.

<table>
<thead>
<tr>
<th>L</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>\text{320} \</td>
</tr>
<tr>
<td>1</td>
<td>CRN_{0:63}</td>
</tr>
<tr>
<td>2</td>
<td>RRN_{0:63}</td>
</tr>
<tr>
<td>3</td>
<td>reserved</td>
</tr>
</tbody>
</table>

Format above is for non-error conditions. 0xFFFFFFFF_FFFFFFFF for error conditions. CRN = conditioned random number RRN = raw random number

A raw random number is unconditioned noise source output. A conditioned random number has been processed by hardware to reduce bias.

Special Registers Altered:
none

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 \textit{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.

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.

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.
### 3.3.9.1 64-bit Fixed-Point Arithmetic Instructions

#### Multiply Low Doubleword

**XO-form**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Function</th>
<th>OE</th>
<th>RC</th>
</tr>
</thead>
<tbody>
<tr>
<td>mulld</td>
<td>RT,RA,RB</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>mulld.</td>
<td>RT,RA,RB</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>mulldo</td>
<td>RT,RA,RB</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>mulldo.</td>
<td>RT,RA,RB</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

$$\prod_{0:127}^{} \leftarrow (RA) \times (RB)$$

$$\text{RT} \leftarrow \prod_{64:127}^{}$$

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**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Function</th>
<th>RC</th>
</tr>
</thead>
<tbody>
<tr>
<td>mulhd</td>
<td>RT,RA,RB</td>
<td>0</td>
</tr>
<tr>
<td>mulhd.</td>
<td>RT,RA,RB</td>
<td>1</td>
</tr>
</tbody>
</table>

$$\prod_{0,127}^{} \leftarrow (RA) \times (RB)$$

$$\text{RT} \leftarrow \prod_{64:127}^{}$$

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**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Function</th>
<th>RC</th>
</tr>
</thead>
<tbody>
<tr>
<td>mulhdu</td>
<td>RT,RA,RB</td>
<td>0</td>
</tr>
<tr>
<td>mulhdu.</td>
<td>RT,RA,RB</td>
<td>1</td>
</tr>
</tbody>
</table>

$$\prod_{0,127}^{} \leftarrow (RA) \times (RB)$$

$$\text{RT} \leftarrow \prod_{64:127}^{}$$

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

\texttt{maddhd RT,RA,RB,RC}

\begin{center}
\begin{tabular}{|c|c|c|c|c|c|}
\hline
4 & 6 & 11 & 16 & 21 & 48 \hline
\end{tabular}
\end{center}

\texttt{prod}_{0:127} \leftarrow (RA) \times (RB) \quad \texttt{sum}_{0:127} \leftarrow \texttt{prod} + \texttt{EXTS}(RC) \quad \texttt{RT} \leftarrow \texttt{sum}_{0:63}

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 RT.

All three operands and the result are interpreted as signed integers.

Special Registers Altered:
None

Multiply-Add High Doubleword Unsigned VA-form

\texttt{maddhdu RT,RA,RB,RC}

\begin{center}
\begin{tabular}{|c|c|c|c|c|c|}
\hline
4 & 6 & 11 & 16 & 21 & 49 \hline
\end{tabular}
\end{center}

\texttt{prod}_{0:127} \leftarrow (RA) \times (RB) \quad \texttt{sum}_{0:127} \leftarrow \texttt{prod} + \texttt{EXTZ}(RC) \quad \texttt{RT} \leftarrow \texttt{sum}_{0:63}

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 RT.

All three operands and the result are interpreted as unsigned integers.

Special Registers Altered:
None

Multiply-Add Low Doubleword VA-form

\texttt{maddld RT,RA,RB,RC}

\begin{center}
\begin{tabular}{|c|c|c|c|c|c|}
\hline
4 & 6 & 11 & 16 & 21 & 51 \hline
\end{tabular}
\end{center}

\texttt{prod}_{0:127} \leftarrow (RA) \times (RB) \quad \texttt{sum}_{0:127} \leftarrow \texttt{prod} + \texttt{EXTS}(RC) \quad \texttt{RT} \leftarrow \texttt{sum}_{64:127}

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 RT.

All three operands and the result are interpreted as signed integers.

Special Registers Altered:
None
### Divide Doubleword XO-form

<table>
<thead>
<tr>
<th>Instruction</th>
<th>RT, RA, RB</th>
<th>(OE=0) (Rc=0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>divd</td>
<td></td>
<td></td>
</tr>
<tr>
<td>divd.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>divdo</td>
<td></td>
<td></td>
</tr>
<tr>
<td>divdo.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Divide Doubleword Unsigned XO-form

<table>
<thead>
<tr>
<th>Instruction</th>
<th>RT, RA, RB</th>
<th>(OE=0) (Rc=0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>divdu</td>
<td></td>
<td></td>
</tr>
<tr>
<td>divdu.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>divduo</td>
<td></td>
<td></td>
</tr>
<tr>
<td>divduo.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### 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) \(= -1\).

\[
\begin{align*}
divd & \quad RT, RA, RB \quad \# RT = quotient \\
mulld & \quad RT, RA, RB \quad \# RT = quotient \times divisor \\
subf & \quad RT, RA, RA \quad \# RT = remainder
\end{align*}
\]

#### Special Registers Altered

- CR0, SO, OV, OV32 (if \(Rc=1\))
- CR0, SO, OV, OV32 (if \(OE=1\))

#### Programming Note

The 64-bit unsigned remainder of dividing (RA) by (RB) can be computed as follows.

\[
\begin{align*}
divdu & \quad RT, RA, RB \quad \# RT = quotient \\
mulld & \quad RT, RA, RB \quad \# RT = quotient \times divisor \\
subf & \quad RT, RA, RA \quad \# RT = remainder
\end{align*}
\]
Divide Doubleword Extended

<table>
<thead>
<tr>
<th>31</th>
<th>22</th>
<th>16</th>
<th>11</th>
<th>6</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>RT</td>
<td>RA</td>
<td>RB</td>
<td>OE</td>
<td>RC</td>
<td>Dividend 0:127 (\equiv RA | 640)</td>
</tr>
<tr>
<td>divisor 0:63 (\equiv RB)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RT \leftarrow \text{dividend} \div \text{divisor}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

<Anything> \div 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 (if RC=1)
- SO
- OV
- OV32 (if OE=1)

Divide Doubleword Extended Unsigned

<table>
<thead>
<tr>
<th>31</th>
<th>22</th>
<th>16</th>
<th>11</th>
<th>6</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>RT</td>
<td>RA</td>
<td>RB</td>
<td>OE</td>
<td>RC</td>
<td>Dividend 0:127 (\equiv RA | 640)</td>
</tr>
<tr>
<td>divisor 0:63 (\equiv RB)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RT \leftarrow \text{dividend} \div \text{divisor}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

<Anything> \div 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 (if RC=1)
- SO
- OV
- OV32 (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{divw[e]u} would be used instead of \texttt{divw[e]u} (and \texttt{cmpld} instead of \texttt{cmplw}, etc.).
Modulo Signed Doubleword X-form
modsd 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></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ \text{dividend} \leftarrow (RA) \]
\[ \text{divisor} \leftarrow (RB) \]
\[ \text{RT} \leftarrow \text{dividend} \% \text{divisor} \]

The 64-bit dividend is (RA). The 64-bit divisor is (RB). The 64-bit remainder 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

\[ \langle \text{anything} \rangle \% 0 \]
\[ 0x8000_0000_0000_0000 \% -1 \]

then the contents of register RT are undefined.

Special Registers Altered:
None

Modulo Unsigned Doubleword X-form
modud 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></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ \text{dividend} \leftarrow (RA) \]
\[ \text{divisor} \leftarrow (RB) \]
\[ \text{RT} \leftarrow \text{dividend} \% \text{divisor} \]

The 64-bit dividend is (RA). The 64-bit divisor is (RB). The 64-bit remainder 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

\[ \langle \text{anything} \rangle \% 0 \]

then the contents of register RT are undefined.

Special Registers Altered:
None
3.3.10 Fixed-Point Compare Instructions

The fixed-point 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 **cmpl** and **cmp**, and unsigned for **cmpli** and **cmpi**.

The L field controls whether the operands are treated as 64-bit or 32-bit quantities, as follows:

<table>
<thead>
<tr>
<th>L</th>
<th>Operand length</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>32-bit operands</td>
</tr>
<tr>
<td>1</td>
<td>64-bit operands</td>
</tr>
</tbody>
</table>

When the operands are treated as 32-bit signed quantities, bit 32 of the register (RA or RB) is the sign bit.

The 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</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LT</td>
<td>(RA) &lt; SI or (RB) (signed comparison)</td>
</tr>
<tr>
<td>1</td>
<td>GT</td>
<td>(RA) &gt; SI or (RB) (signed comparison)</td>
</tr>
<tr>
<td>2</td>
<td>EQ</td>
<td>(RA) = SI, UI, or (RB)</td>
</tr>
<tr>
<td>3</td>
<td>SO</td>
<td>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 Compare instructions. See Appendix C for additional extended mnemonics.
Compare Immediate

\[
\text{cmpi BF}, L, RA, SI
\]

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 \( c \leftarrow 0b001 \)
\( \text{CR4} \times BF + 32:4 \times BF + 35 \leftarrow c \mid \text{XERSO} \)

The contents of register RA ((RA)\(_{32:63}\) 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:

**Extended:**
- \( \text{cmpdi Rx,value} \)
- \( \text{cmpwi cr3,Rx,value} \)

**Equivalent to:**
- \( \text{cmp} 0,1,Rx,value \)
- \( \text{cmp} 3,0,Rx,value \)

Compare

\[
\text{cmp BF}, L, RA, RB
\]

if \( L = 0 \) then \( a \leftarrow \text{EXTS}((RA)_{32:63}) \)
else \( a \leftarrow (RA) \)
if \( a < b \) then \( c \leftarrow 0b100 \)
else if \( a > b \) then \( c \leftarrow 0b010 \)
else \( c \leftarrow 0b001 \)
\( \text{CR4} \times BF + 32:4 \times BF + 35 \leftarrow c \mid \text{XERSO} \)

The contents of register RA ((RA)\(_{32:63}\) if \( L=0 \)) are compared with the contents of register RB ((RB)\(_{32:63}\) if \( L=0 \)), 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:
**Compare Logical Immediate**

<table>
<thead>
<tr>
<th>D-form</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>cmpli</td>
<td>BF, L, RA, UI</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>BF</td>
<td>/</td>
<td>L</td>
<td>10</td>
</tr>
</tbody>
</table>

if $L = 0$ then $a \leftarrow 32_0 || (RA)_{32:63}$
else $a \leftarrow (RA)$
if $a <^u 48_0 || UI$ then $c \leftarrow 0b100$
else if $a >^u 48_0 || UI$ then $c \leftarrow 0b010$
else $c \leftarrow 0b001$
$CR_{4*BF+32:4*BF+35} \leftarrow c || XER_{SO}$

The contents of register RA ($(RA)_{32:63}$ zero-extended to 64 bits if $L=0$) are compared with $48_0 || UI$, 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**:

- **Extended:** cmpli
  - Equivalent to: cmpli 0,1,Rx,value
  - cmpli 3,0,Rx,value

---

**Compare Logical**

**X-form**

<table>
<thead>
<tr>
<th>X-form</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>cmpl</td>
<td>BF, L, RA, RB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>31</td>
<td>BF</td>
<td>/</td>
<td>L</td>
<td>10</td>
</tr>
</tbody>
</table>

if $L = 0$ then $a \leftarrow 32_0 || (RA)_{32:63}$
else $a \leftarrow (RA)$
if $a <^u b$ then $c \leftarrow 0b100$
else if $a >^u b$ then $c \leftarrow 0b010$
else $c \leftarrow 0b001$
$CR_{4*BF+32:4*BF+35} \leftarrow c || XER_{SO}$

The contents of register RA ($(RA)_{32:63}$ if $L=0$) 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**:

- **Extended:** cmpl
  - Equivalent to: cmpl 0,1,Rx,Ry
  - cmpl 3,0,Rx,Ry
### Compare Ranged Byte X-form

- **cmprb BF,L,RA,RB**

<table>
<thead>
<tr>
<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>6</td>
<td>10</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

- 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.

- If L=0, then `in_range ← (src22lo ≤ src1) & (src1 ≤ src22hi)`
  - Else, `in_range ← (src21lo ≤ src1) & (src1 ≤ src21hi) | (src22lo ≤ src1) & (src1 ≤ src22hi)`

- CR<sub>4+BF+32</sub> ← 0b0
- CR<sub>4+BF+33</sub> ← in_range
- CR<sub>4+BF+34</sub> ← 0b0
- CR<sub>4+BF+35</sub> ← 0b0

A single-range compare can be implemented with an `addi` to load the upper and lower bounds in the range, such as `isdigit()`. A combination of `addi-addis` can be used to set up 2 ranges, such as for `isalpha()`.

**Programming Note**

- **cmprb** is useful for implementing character typing functions such as `isalpha()`, `isdigit()`, `isupper()`, and `islower()` that are implemented using one or two range compares of the character.

- A single-range compare can be implemented with an `addi` to load the upper and lower bounds in the range, such as `isdigit()`.

  ```
  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
  ```

  ```
  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,
  ; setting CR field TGT to
  ; indicate in range
  ```

- A combination of `addi-addis` can be used to set up 2 ranges, such as for `isalpha()`.

  ```
  addi rRNG,0,0x7A41       ; 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,
  ; setting CR field TGT to
  ; indicate in range
  ```

**Special Registers Altered:**

- CR field BF
Compare Equal Byte X-form

\texttt{cmpeqb} BF,RA,RB

<table>
<thead>
<tr>
<th>31</th>
<th>BF</th>
<th>RA</th>
<th>RB</th>
<th>224</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>18</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

src ← GPR[RA].bit[56:63]


CR field BF 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

\textbf{Programming Note}

\texttt{cmpeqb} is useful for implementing character typing functions such as \texttt{isspace()} that are implemented by comparing the character to 1 or more values.

A function such as \texttt{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 \texttt{cmpeb} to compare the subject character to those 6 values to determine if any match occurs.

\texttt{ld x rSPC, WS_CHARS ; rSPC = 0x09_0A_0B_0C_20}
\texttt{cmpeqb 2, cr1, rCHAR, rSPC ; perform match compare on}

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

\[
\text{twi} \quad \text{TO,RA,SI}
\]

\[
\begin{array}{cccc}
\text{D-form} & \text{X-form} \\
\text{0} & \text{3} & \text{5} & \text{11} & \text{16} & \text{31} & \text{31} & \text{5} & \text{11} & \text{16} & \text{21} & 4 & 31 \\
\end{array}
\]

a \leftarrow \text{EXTS}((\text{RA})_{32:63})
if (a < \text{EXTS(SI)}) & \text{TO}^0 \text{ then TRAP}
if (a > \text{EXTS(SI)}) & \text{TO}^1 \text{ then TRAP}
if (a = \text{EXTS(SI)}) & \text{TO}^2 \text{ then TRAP}
if (a <u \text{EXTS(SI)}) & \text{TO}^3 \text{ then TRAP}
if (a >u \text{EXTS(SI)}) & \text{TO}^4 \text{ then TRAP}

The contents of \(\text{RA}_{32:63}\) 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: Equivalent to:
twgli Rx,value twi 8,Rx,value
twle i Rx,value twi 6,Rx,value

Trap Word

\[
\text{tw} \quad \text{TO,RA,RB}
\]

\[
\begin{array}{cccc}
\text{X-form} & \text{D-form} \\
\text{0} & \text{31} & \text{5} & \text{11} & \text{16} & 21 & 4 & 31 \\
\end{array}
\]

a \leftarrow \text{EXTS}((\text{RA})_{32:63})
b \leftarrow \text{EXTS}((\text{RB})_{32:63})
if (a < b) & \text{TO}^0 \text{ then TRAP}
if (a > b) & \text{TO}^1 \text{ then TRAP}
if (a = b) & \text{TO}^2 \text{ then TRAP}
if (a <u b) & \text{TO}^3 \text{ then TRAP}
if (a >u b) & \text{TO}^4 \text{ then TRAP}

The contents of \(\text{RA}_{32:63}\) are compared with the contents of \(\text{RB}_{32:63}\). 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: Equivalent to:
tweq Rx,Ry tw 4,Rx,Ry
twje Rx,Ry tw 5,Rx,Ry
trap tw 31,0,0
3.3.11.1 64-bit Fixed-Point Trap Instructions

**Trap Doubleword Immediate**  \( D \)-form

\[
\text{tdi} \quad \text{TO,RA,SI}
\]

\[
\begin{array}{cccc}
2 & 6 & 11 & 16 & 31 \\
0 & & & & \\
\end{array}
\]

\[a \leftarrow (RA)\]
\[b \leftarrow \text{EXTS(SI)}\]

- if \((a < b) \& \& \text{TO}_0\) then TRAP
- if \((a > b) \& \& \text{TO}_1\) then TRAP
- if \((a = b) \& \& \text{TO}_2\) then TRAP
- if \((a < u b) \& \& \text{TO}_3\) then TRAP
- if \((a > u b) \& \& \text{TO}_4\) then TRAP

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**:

- Extended: \( \text{tdlti Rx,value tdi 16,Rx,value} \)
- Extended: \( \text{tdnei Rx,value tdi 24,Rx,value} \)

**Trap Doubleword**  \( X \)-form

\[
\text{td} \quad \text{TO,RA,RB}
\]

\[
\begin{array}{ccccccc}
31 & 6 & 11 & 16 & 21 & 31 \\
0 & & & & & \\
\end{array}
\]

\[a \leftarrow (RA)\]
\[b \leftarrow (RB)\]

- if \((a < b) \& \& \text{TO}_0\) then TRAP
- if \((a > b) \& \& \text{TO}_1\) then TRAP
- if \((a = b) \& \& \text{TO}_2\) then TRAP
- if \((a < u b) \& \& \text{TO}_3\) then TRAP
- if \((a > u b) \& \& \text{TO}_4\) then TRAP

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**:

- Extended: \( \text{tdge Rx,Ry tdi 12,Rx,Ry} \)

3.3.12 Fixed-Point Select

**Integer Select**  \( A \)-form

\[
isel \quad \text{RT,RA,RB,BC}
\]

\[
\begin{array}{cccccccc}
31 & 6 & 11 & 16 & 21 & 26 & 15 & 31 \\
0 & & & & & & & \\
\end{array}
\]

if \(\text{RA}=0\) then \(a \leftarrow (RA)\)
if \(\text{CR}_{BC+32}=1\) then \(\text{RT} \leftarrow a\)
else \(\text{RT} \leftarrow \text{RB}\)

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**:

- Extended: \( \text{isellt Rx,Ry,Rz isel Rx,Ry,Rz,0} \)
- Extended: \( \text{iselgt Rx,Ry,Rz isel Rx,Ry,Rz,1} \)
- Extended: \( \text{isелеq Rx,Ry,Rz isel Rx,Ry,Rz,2} \)
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 \( \text{andi} \) and \( \text{andis} \), set the first three bits of CR Field 0 as described in Section 3.3.8, “Other Fixed-Point Instructions” on page 66. The Logical instructions do not change the SO, 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 \( \text{OR Immediate} \) 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 \( \text{XOR Immediate} \) 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 \( \text{OR} \) and \( \text{NOR} \) 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 791 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**

\( \text{andi} \) \( \text{RA,RS,UI} \)

<table>
<thead>
<tr>
<th>( 28 )</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>

\( RA \leftarrow \text{RS} \land (480 \| UI) \)

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**

\( \text{andis} \) \( \text{RA,RS,UI} \)

<table>
<thead>
<tr>
<th>( 29 )</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>

\( RA \leftarrow \text{RS} \land (320 \| UI \| 160) \)

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**

\( \text{ori} \) \( \text{RA,RS,UI} \)

<table>
<thead>
<tr>
<th>( 24 )</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>

\( RA \leftarrow (\text{RS}) \lor (480 \| UI) \)

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:

\( \text{ori} 0,0,0 \)

**Special Registers Altered:**

None

**Extended Mnemonics:**

Example of extended mnemonics for \( \text{OR Immediate} \):

\[ \text{Extended:} \quad \text{Equivalent to:} \]

\[ \text{no-op} \quad \text{ori} 0,0,0 \]
### OR Immediate Shifted

**D-form**

<table>
<thead>
<tr>
<th>oriS</th>
<th>RA,RS,UI</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>25 6</td>
</tr>
</tbody>
</table>

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 Shifted

**D-form**

<table>
<thead>
<tr>
<th>xorS</th>
<th>RA,RS,UI</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>27 6</td>
</tr>
</tbody>
</table>

The contents of register RS are XORed with \(480 \| UI \| 160\) and the result is placed into register RA.

**Special Registers Altered:**

None

### XOR Immediate

**D-form**

<table>
<thead>
<tr>
<th>xori</th>
<th>RA,RS,UI</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>25 6</td>
</tr>
</tbody>
</table>

The contents of register RS are XORed with \(480 \| UI\) and the result is placed into register RA.

**Extended Mnemonics:**

Example of extended mnemonics for XOR Immediate:

<table>
<thead>
<tr>
<th>Extended</th>
<th>Equivalent to:</th>
</tr>
</thead>
<tbody>
<tr>
<td>xnop</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.
### AND

<table>
<thead>
<tr>
<th>X-form</th>
<th>RA,RS,RB</th>
<th>(Rc=0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>and.</td>
<td>RA,RS,RB</td>
<td>(Rc=1)</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.

Some forms of and Rx, Rx, Rx provide special functions; see Section 9.3 of Book III.

**Special Registers Altered:**

- CR0 (if Rc=1)

### XOR

<table>
<thead>
<tr>
<th>X-form</th>
<th>RA,RS,RB</th>
<th>(Rc=0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>xor.</td>
<td>RA,RS,RB</td>
<td>(Rc=1)</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:**

- CR0 (if Rc=1)

### NAND

<table>
<thead>
<tr>
<th>X-form</th>
<th>RA,RS,RB</th>
<th>(Rc=0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>nand.</td>
<td>RA,RS,RB</td>
<td>(Rc=1)</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

<table>
<thead>
<tr>
<th>X-form</th>
<th>RA,RS,RB</th>
<th>(Rc=0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>or.</td>
<td>RA,RS,RB</td>
<td>(Rc=1)</td>
</tr>
</tbody>
</table>

RA ← (RS) | (RB)

The contents of register RS 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:**

- CR0 (if Rc=1)

**Extended Mnemonics:**

Example of extended mnemonics for OR:

**Extended:**

<table>
<thead>
<tr>
<th>Equivalent to:</th>
</tr>
</thead>
<tbody>
<tr>
<td>mr Rx,Ry</td>
</tr>
<tr>
<td>or Rx,Ry,Ry</td>
</tr>
</tbody>
</table>

---

### Extended: Equivalent to:

<table>
<thead>
<tr>
<th>Extended:</th>
</tr>
</thead>
<tbody>
<tr>
<td>CR0</td>
</tr>
</tbody>
</table>

---

**Reference:**

Chapter 3 of Book III.
### Chapter 3. Fixed-Point Facility

#### Fixed-Point Facility

**NOR**

- **NOR** RA, RS, RB (Rc=0)
- **NOR** RA, RS, RB (Rc=1)

**X-form**

```
0 31 RS RA RB 124 RC
```

- RA ← ¬((RS) | (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:**

- CR0 (if Rc=1)

#### Equivalent

**Equivalent** RA, RS, RB (Rc=0)

**X-form**

```
0 31 RS RA RB 284 RC
```

- RA ← (RS) = (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:**

- CR0 (if Rc=1)

#### Extended Mnemonics:

Example of extended mnemonics for **NOR**:

- Extended: **not Rx,Ry**
- Equivalent to: **nor Rx,Ry,Ry**

### AND with Complement

- **AND with Complement** RA, RS, RB (Rc=0)
- **AND with Complement** RA, RS, RB (Rc=1)

**X-form**

```
0 31 RS RA RB 60 RC
```

- RA ← (RS) & ¬(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:**

- CR0 (if Rc=1)

### OR with Complement

- **OR with Complement** RA, RS, RB (Rc=0)
- **OR with Complement** RA, RS, RB (Rc=1)

**X-form**

```
0 31 RS RA RB 412 RC
```

- RA ← (RS) | ¬(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:**

- CR0 (if Rc=1)
### Extend Sign Byte

**X-form**  
extsb  RA,RS  (Rc=0)  
extsb. RA,RS  (Rc=1)

<table>
<thead>
<tr>
<th>31</th>
<th>16</th>
<th>11</th>
<th>6</th>
<th>55</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

s ← (RS)_{56}^{63}  
RA_{56:63} ← (RS)_{56:63}  
RA_{0:55} ← 56_{8}

(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)

### Count Leading Zeros Word

**X-form**  
cntzgw  RA,RS  (Rc=0)  
cntzg.w  RA,RS  (Rc=1)

<table>
<thead>
<tr>
<th>31</th>
<th>16</th>
<th>11</th>
<th>6</th>
<th>26</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

n ← 32  
do while n < 64  
  if (RS)_n = 1 then leave  
  n ← n + 1  
RA ← 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.  

If Rc is equal to 1, CR field 0 is set to reflect the result.  

**Special Registers Altered:**  
CR0 (if Rc=1)

### Count Trailing Zeros Word

**X-form**  
cnttzgw  RA,RS  (Rc=0)  
cnttzg.w  RA,RS  (Rc=1)

<table>
<thead>
<tr>
<th>31</th>
<th>16</th>
<th>11</th>
<th>6</th>
<th>31</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

n ← 0  
do while n < 32  
  if (RS)_{63-n} = 0b1 then leave  
  n ← n + 1  
RA ← 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.  

If Rc is equal to 1, CR field 0 is set to reflect the result.  

**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.
**Compare Bytes**

`cmpb` RA, RS, RB

<table>
<thead>
<tr>
<th>X-form</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
</tr>
</tbody>
</table>

```plaintext
do n = 0 to 7
  if RS8xn:8xn+7 = RB8xn:8xn+7 then
    RA8xn:8xn+7 = 1
  else
    RA8xn:8xn+7 = 0
```

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

**Population Count Bytes**

`popcntb` RA, RS

<table>
<thead>
<tr>
<th>X-form</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
</tr>
</tbody>
</table>

```plaintext
do i = 0 to 7
  n = 0
  do j = 0 to 7
    if (RS)(i*8)+j = 1 then
      n = n+1
      RA(i*8):(i*8)+7 = n
  end do
```

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**

`popcntw` RA, RS

<table>
<thead>
<tr>
<th>X-form</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
</tr>
</tbody>
</table>

```plaintext
do i = 0 to 1
  n = 0
  do j = 0 to 31
    if (RS)(i*32)+j = 1 then
      n = n+1
      RA(i*32):(i*32)+31 = n
  end do
```

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 Doubleword: X-form

prtyd RA, RS

<table>
<thead>
<tr>
<th>31</th>
<th>RS</th>
<th>RA</th>
<th>///</th>
<th>186</th>
<th>/</th>
</tr>
</thead>
</table>

s ← 0
\[\text{do } i = 0 \text{ to } 7\]
\[s \leftarrow s / (RS)_{i} \mod 8 + 7\]
RA ← \[63, 0 \mid s\]

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

Parity Word: X-form

prtyw RA, RS

<table>
<thead>
<tr>
<th>31</th>
<th>RS</th>
<th>RA</th>
<th>///</th>
<th>154</th>
<th>/</th>
</tr>
</thead>
</table>

s ← 0
\[\text{do } i = 0 \text{ to } 3\]
\[s \leftarrow s / (RS)_{i} \mod 8 + 7\]
\[\text{do } i = 4 \text{ to } 7\]
\[t \leftarrow t / (RS)_{i} \mod 8 + 7\]
RA\[0:31\] ← \[31, 0 \mid s\]
RA\[32:63\] ← \[31, 0 \mid t\]

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 RA\[0:31\]; otherwise the value 0 is placed into RA\[0:31\]. 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 RA\[32:63\]; otherwise the value 0 is placed into RA\[32:63\].

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.

\[\text{popcntb } RA, RS\]
\[\text{prtyw } RA, RA\]

The parity of (RS) can be computed as follows.

\[\text{popcntb } RA, RS\]
\[\text{prtyd } RA, RA\]
### 3.3.13.1 64-bit Fixed-Point Logical Instructions

#### Extend Sign Word

<table>
<thead>
<tr>
<th>X-form</th>
<th>Extsw</th>
<th>RA, RS&lt;br&gt;(Rc=0)</th>
<th>Extsw.</th>
<th>RA, RS&lt;br&gt;(Rc=1)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>31</td>
<td>RS</td>
<td>RA</td>
<td>///</td>
</tr>
</tbody>
</table>

\[
s \leftarrow (RS)_{32}
\]

\[
RA_{32:63} \leftarrow (RS)_{32:63}
\]

\[
RA_{0:31} \leftarrow 32 s
\]

(RS)\(_{32:63}\) are placed into RA\(_{32:63}\). RA\(_{0:31}\) are filled with a copy of (RS)\(_{32}\).

**Special Registers Altered:**

- CR0 (if Rc=1)

#### Population Count Doubleword

<table>
<thead>
<tr>
<th>X-form</th>
<th>Popcntd</th>
<th>RA, RS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>31</td>
<td>RS</td>
</tr>
</tbody>
</table>

\[
n \leftarrow 0
\]

\[
do i = 0 to 63
\]

\[
if (RS)_i = 1 then
\]

\[
n \leftarrow n + 1
\]

\[
RA \leftarrow n
\]

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

#### Count Leading Zeros Doubleword

<table>
<thead>
<tr>
<th>X-form</th>
<th>Cntlzd</th>
<th>RA, RS&lt;br&gt;(Rc=0)</th>
<th>Cntlzd.</th>
<th>RA, RS&lt;br&gt;(Rc=1)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>31</td>
<td>RS</td>
<td>RA</td>
<td>///</td>
</tr>
</tbody>
</table>

\[
n \leftarrow 0
\]

\[
do while n < 64
\]

\[
if (RS)_n = 1 then leave
\]

\[
n \leftarrow n + 1
\]

\[
RA \leftarrow n
\]

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

<table>
<thead>
<tr>
<th>X-form</th>
<th>Cntlzd</th>
<th>RA, RS&lt;br&gt;(Rc=0)</th>
<th>Cntlzd.</th>
<th>RA, RS&lt;br&gt;(Rc=1)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>31</td>
<td>RS</td>
<td>RA</td>
<td>///</td>
</tr>
</tbody>
</table>

\[
n \leftarrow 0
\]

\[
do while n < 64
\]

\[
if (RS)_{63-n} = 0b1 then leave
\]

\[
n \leftarrow n + 1
\]

\[
RA \leftarrow EXTZ64(n)
\]

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:**

- CR0 (if Rc=1)
**Bit Permute Doubleword**

For i = 0 to 7

\[
\text{index} \leftarrow (\text{RS})_8 \times i; 8 \times i + 7
\]

If index < 64

\[
\text{perm}_i \leftarrow (\text{RB})_{\text{index}}
\]

else
\[
\text{perm}_i \leftarrow 0
\]

\[
\text{RA} \leftarrow 560 \big| \big| \text{perm}_{0:7}
\]

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.

```
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
```
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 rotate64 or ROTL64, the value rotated is the given 64-bit value. The rotate64 operation is used to rotate a given 64-bit quantity.

For the second type, denoted rotate32 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 rotate32 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:

\[
\begin{align*}
\text{if } mstart &\leq mstop \text{ then } \\
& \quad \text{mask}_{mstart:mstop} = \text{ones} \\
& \quad \text{mask}_{\text{all other bits}} = \text{zeros} \\
\text{else } & \\
& \quad \text{mask}_{mstart:63} = \text{ones} \\
& \quad \text{mask}_{0:mstop} = \text{ones} \\
& \quad \text{mask}_{\text{all other bits}} = \text{zeros}
\end{align*}
\]

There is no way to specify an all-zero mask.

For instructions that use the rotate32 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 66. 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 791 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.


Rotate Left Word Immediate then AND with Mask M-form

\[
\text{rlwinm} \quad \text{RA,RS,SH,MB,ME} \quad \text{(Rc=0)} \\
\text{rlwinm} \quad \text{RA,RS,SH,MB,ME} \quad \text{(Rc=1)} \\
\]

\[
\begin{array}{ccccccc}
0 & 21 & 6 & 11 & 16 & 21 & 26 & 31 \\
\text{RS} & \text{RA} & \text{SH} & \text{MB} & \text{ME} & \text{Rc} \\
\end{array}
\]

\[
\begin{align*}
n & \leftarrow \text{SH} \\
r & \leftarrow \text{ROTL}_{32}(\text{(RS)}_{32:63}, n) \\
m & \leftarrow \text{MASK(MB+32, ME+32)} \\
\text{RA} & \leftarrow r \land m
\end{align*}
\]

The contents of register RS are rotated 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:

\[
\text{CR0} \quad \text{(if Rc=1)}
\]

Extended Mnemonics:

Examples of extended mnemonics for Rotate Left Word Immediate then AND with Mask:

\[
\begin{align*}
\text{Extended:} & \quad \text{Equivalent to:} \\
\text{exltwi Rx,Ry,n,b} & \quad \text{rlwinm Rx,Ry,b,0,n-1} \\
\text{srwi Rx,Ry,n} & \quad \text{rlwinm Rx,Ry,32-n,n,31} \\
\text{clrwi Rx,Ry,n} & \quad \text{rlwinm Rx,Ry,0,0,31-n}
\end{align*}
\]

---

Programming Note

Let RSL represent the low-order 32 bits of register RS, with the bits numbered from 0 through 31.

\text{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=31. 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=32-n, and ME=31. 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 the contents of a register and then shift the result left by n bits, by setting SH=n, MB=b-n, and ME=31. 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 all of these uses; see Appendix C, “Assembler Extended Mnemonics” on page 791.
Rotate Left Word then AND with Mask

\[\text{M-form} \]

\[
\begin{array}{cccccc}
\text{rlwmm} & \text{RA,RS,RB,MB,ME} \\
\text{rlwmm} & \text{RA,RS,RB,MB,ME} \\
\end{array}
\]

\[\text{(Rc}=0)\]

\[\text{RA,RS,RB,MB,ME} \]

\[\text{(Rc}=1)\]

\[
\begin{array}{cccccc}
\text{n} & \rightarrow & (\text{RB})_{59:63} \\
\text{r} & \rightarrow & \text{ROTL}_{32}(\text{RS})_{32:63},\ n \\
m & \rightarrow & \text{MASK}(\text{MB+32},\ ME+32) \\
\text{RA} & \rightarrow & r \& m \\
\end{array}
\]

The contents of register RS are rotated 32 left the number of bits specified by \((\text{RB})_{59:63}\). 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:

\[\text{CR0} \]

Extended Mnemonics:

Example of extended mnemonics for Rotate Left Word then AND with Mask:

Extended: \(\text{rotlw Rx,Ry,Rz}\)

Equivalent to: \(\text{rlwmm Rx,Ry,Rz,0,31}\)

Programming Note

Let RSL represent the low-order 32 bits of register RS, with the bits numbered from 0 through 31.

\(\text{rlwmm}\) can be used to extract an n-bit field that starts at variable 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 \(\text{RB}_{59:63}=b+n\), MB=32-n, and ME=31. It can be used to extract an n-bit field that starts at variable 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 \(\text{RB}_{59:63}=b\), 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 variable n bits, by setting \(\text{RB}_{59:63}=n\) (32-n), MB=0, and ME=31.

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 791.

Rotate Left Word Immediate then Mask Insert

\[\text{M-form} \]

\[
\begin{array}{cccccc}
\text{rlwimi} & \text{RA,RS,SH,MB,ME} \\
\text{rlwimi} & \text{RA,RS,SH,MB,ME} \\
\end{array}
\]

\[\text{(Rc}=0)\]

\[\text{RA,RS,SH,MB,ME} \]

\[\text{(Rc}=1)\]

\[
\begin{array}{cccccc}
\text{n} & \rightarrow & \text{SH} \\
r & \rightarrow & \text{ROTL}_{32}(\text{RS})_{32:63},\ n \\
m & \rightarrow & \text{MASK}(\text{MB+32},\ ME+32) \\
\text{RA} & \rightarrow & r \& m | (\text{RA}) \& \neg m \\
\end{array}
\]

The contents of register RS are rotated 32 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 inserted into register RA under control of the generated mask.

Special Registers Altered:

\[\text{CR0} \]

Extended Mnemonics:

Example of extended mnemonics for Rotate Left Word Immediate then Mask Insert:

Extended: \(\text{inslw Rx,Ry,n,b}\)

Equivalent to: \(\text{rlwimi Rx,Ry,32-b,b,b+n-1}\)

Programming Note

Let RAL represent the low-order 32 bits of register RA, with the bits numbered from 0 through 31.

\(\text{rlwimi}\) can be used to insert an n-bit field that is left-justified in the low-order 32 bits of register RS, into RAL starting at bit position b, by setting \(\text{SH}=32-b, \text{MB}=b,\) and \(\text{ME}=(b+n)-1\). It can be used to insert an n-bit field that is right-justified in the low-order 32 bits of register RS, into RAL starting at bit position b, by setting \(\text{SH}=32-(b+n), \text{MB}=b,\) and \(\text{ME}=(b+n)-1\).

Extended mnemonics are provided for both of these uses; see Appendix C, “Assembler Extended Mnemonics” on page 791.
3.3.14.1.1 64-bit Fixed-Point Rotate Instructions

**Rotate Left Doubleword Immediate then Clear Left**

<table>
<thead>
<tr>
<th></th>
<th>RA, RS, SH, MB</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>rldicl</td>
<td></td>
<td>(Rc=0)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rldicl</td>
<td></td>
<td>(Rc=1)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RS</th>
<th>RA</th>
<th>sh</th>
<th>mb</th>
<th>0</th>
<th>sh</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>27</td>
<td>30</td>
</tr>
</tbody>
</table>

\[ n \leftarrow sh_0 \parallel sh_{0:4} \]
\[ r \leftarrow \text{ROTL}_{64}(RS), n \]
\[ b \leftarrow mb_5 \parallel mb_{0:4} \]
\[ m \leftarrow \text{MASK}(b, 63) \]
\[ RA \leftarrow r \& m \]

The contents of register RS are rotated \( 64 \) left \( SH \) bits. 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:**

Examples of extended mnemonics for **Rotate Left Doubleword Immediate then Clear Left**:

<table>
<thead>
<tr>
<th>Extended</th>
<th>Equivalent to:</th>
</tr>
</thead>
<tbody>
<tr>
<td>extrdi</td>
<td>rldicl Rx,Ry,b+n,64-n</td>
</tr>
<tr>
<td>srdi</td>
<td>rldicl Rx,Ry,64-n</td>
</tr>
<tr>
<td>clrrdi</td>
<td>rldicl Rx,Ry,0,n</td>
</tr>
</tbody>
</table>

**Programming Note**

**rldicl** 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 791.

**Rotate Left Doubleword Immediate then Clear Right**

<table>
<thead>
<tr>
<th></th>
<th>RA, RS, SH, ME</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>rldicr</td>
<td></td>
<td>(Rc=0)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rldicr</td>
<td></td>
<td>(Rc=1)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RS</th>
<th>RA</th>
<th>sh</th>
<th>me</th>
<th>1</th>
<th>sh</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>27</td>
<td>30</td>
</tr>
</tbody>
</table>

\[ n \leftarrow sh_5 \parallel sh_{0:4} \]
\[ r \leftarrow \text{ROTL}_{64}(RS), n \]
\[ e \leftarrow me_5 \parallel me_{0:4} \]
\[ m \leftarrow \text{MASK}(0, e) \]
\[ RA \leftarrow r \& m \]

The contents of register RS are rotated \( 64 \) left \( SH \) bits. A mask is generated having 1-bits from bit 0 through 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) |

**Extended Mnemonics:**

Examples of extended mnemonics for **Rotate Left Doubleword Immediate then Clear Right**:

<table>
<thead>
<tr>
<th>Extended</th>
<th>Equivalent to:</th>
</tr>
</thead>
<tbody>
<tr>
<td>extldi</td>
<td>rldicr Rx,Ry,b-1</td>
</tr>
<tr>
<td>sldi</td>
<td>rldicr Rx,Ry,n,63-n</td>
</tr>
<tr>
<td>clrrdi</td>
<td>rldicr Rx,Ry,0,63-n</td>
</tr>
</tbody>
</table>

**Programming Note**

**rldicr** 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 **rldicl**); see Appendix C, "Assembler Extended Mnemonics" on page 791.
Rotate Left Doubleword Immediate then Clear

**MD-form**

rldic RA, RS, SH, MB (Rc=0)
rldic RA, RS, SH, MB (Rc=1)

<table>
<thead>
<tr>
<th>30</th>
<th>8</th>
<th>16</th>
<th>21</th>
<th>27</th>
<th>30</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 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:

- Extended: c1rldi Rx, Ry, b, n
- Equivalent to: rldic 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-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 `rldic`); see Appendix C, “Assembler Extended Mnemonics” on page 791.

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>8</th>
<th>16</th>
<th>21</th>
<th>27</th>
<th>30</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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:

- Extended: rotld Rx, Ry, Rz
- Equivalent to: 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 RB58:63=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 RB58:63=n (64-n) and MB=0.

Extended mnemonics are provided for some of these uses; see Appendix C, “Assembler Extended Mnemonics” on page 791.
**Rotate Left Doubleword then Clear Right**

<table>
<thead>
<tr>
<th>MDS-form</th>
<th>MD-form</th>
</tr>
</thead>
<tbody>
<tr>
<td>rldcr</td>
<td>rldimi</td>
</tr>
<tr>
<td>RA,RS,RB,ME</td>
<td>RA,RS,SH,MB</td>
</tr>
</tbody>
</table>

(Rc=0)

(Rc=1)

<table>
<thead>
<tr>
<th>30</th>
<th>31</th>
<th>27</th>
<th>21</th>
<th>16</th>
<th>11</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>b</td>
<td>me</td>
<td>RB</td>
<td>RA</td>
<td>RS</td>
<td></td>
</tr>
</tbody>
</table>

\[
n \leftarrow (RB)_{58:63}
\]

\[
r \leftarrow \text{ROTL}_{64}(\text{RS}, n)
\]

\[
e \leftarrow \text{MASK}(0, e)
\]

\[
m \leftarrow r \& m
\]

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 0 through 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 791.

**Rotate Left Doubleword Immediate then Mask Insert**

<table>
<thead>
<tr>
<th>MDS-form</th>
<th>MD-form</th>
</tr>
</thead>
<tbody>
<tr>
<td>rldcr</td>
<td>rldimi</td>
</tr>
<tr>
<td>RA,RS,RB,ME</td>
<td>RA,RS,SH,MB</td>
</tr>
</tbody>
</table>

(Rc=0)

(Rc=1)

<table>
<thead>
<tr>
<th>30</th>
<th>31</th>
<th>27</th>
<th>21</th>
<th>16</th>
<th>11</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>b</td>
<td>sh</td>
<td>mb</td>
<td>RA</td>
<td>RS</td>
<td></td>
</tr>
</tbody>
</table>

\[
n \leftarrow sh_{5} \gg sh_{0:4}
\]

\[
r \leftarrow \text{ROTL}_{64}(\text{RS}, n)
\]

\[
b \leftarrow mb_{5} \gg mb_{0:4}
\]

\[
m \leftarrow \text{MASK}(b, \neg n)
\]

\[
RA \leftarrow R \& m \| (RA) \& \neg m
\]

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:**

\[
\text{insrdi} \quad Rx,Ry,n,b
\]

**Equivalent to:**

\[
rldimi \quad Rx,Ry,64−(b+n),b
\]

**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 791.
### 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 791 for additional extended mnemonics.

#### Shift Left Word

<table>
<thead>
<tr>
<th>X-form</th>
</tr>
</thead>
<tbody>
<tr>
<td>slw</td>
</tr>
<tr>
<td>slw.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>X-form</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
</tr>
<tr>
<td>r</td>
</tr>
<tr>
<td>if (RB)_58 = 0 then</td>
</tr>
<tr>
<td>m</td>
</tr>
<tr>
<td>else m ← 640</td>
</tr>
<tr>
<td>RA ← r &amp; m</td>
</tr>
</tbody>
</table>

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:

CR0 (if Rc=1)

---

#### Shift Right Word

<table>
<thead>
<tr>
<th>X-form</th>
</tr>
</thead>
<tbody>
<tr>
<td>srw</td>
</tr>
<tr>
<td>srw.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>X-form</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
</tr>
<tr>
<td>r</td>
</tr>
<tr>
<td>if (RB)_58 = 0 then</td>
</tr>
<tr>
<td>m</td>
</tr>
<tr>
<td>else m ← 640</td>
</tr>
<tr>
<td>RA ← r &amp; m</td>
</tr>
</tbody>
</table>

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:

CR0 (if Rc=1)
**Shift Right Algebraic Word Immediate X-form**

srawi: RA, RS, SH (Rc=0)
srawi: RA, RS, SH (Rc=1)

<table>
<thead>
<tr>
<th>31</th>
<th>6</th>
<th>16</th>
<th>21</th>
<th>824</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>

n ← SH
r ← ROTL32((RS)32:63, 64-n)
m ← MASK(n+32, 63)
s ← (RS)32
RA ← r&m | (64s)&¬m
carry ← s & ((r&¬m)32:63 != 0)
CA ← carry
CA32 ← 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 RA32:63.

**Special Registers Altered:**
- CA
- CA32
- CR0 (if Rc=1)

**Shift Right Algebraic Word X-form**

sraw: RA, RS, RB (Rc=0)
sraw: RA, RS, RB (Rc=1)

<table>
<thead>
<tr>
<th>31</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>792</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>
<td>31</td>
</tr>
</tbody>
</table>

n ← (RB)59:63
r ← ROTL32((RS)32:63, 64-n)
if (RB)58 = 0 then
  m ← MASK(n+32, 63)
else m ← 640
s ← (RS)32
RA ← r&m | (64s)&¬m
carry ← s & ((r&¬m)32:63 != 0)
CA ← carry
CA32 ← carry

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. Bit 32 of RS is replicated to fill the vacated positions on the left. The 32-bit result is placed into RA32:63.

**Special Registers Altered:**
- CA
- CA32
- CR0 (if Rc=1)
### 3.3.14.2.1 64-bit Fixed-Point Shift Instructions

#### Shift Left Doubleword

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Source</th>
<th>Destination</th>
<th>X-form</th>
</tr>
</thead>
<tbody>
<tr>
<td>sld</td>
<td>RA,RS,RB</td>
<td>RA</td>
<td>(Rc=0)</td>
</tr>
<tr>
<td>sld.</td>
<td>RA,RS,RB</td>
<td>RA</td>
<td>(Rc=1)</td>
</tr>
</tbody>
</table>

\[
\begin{array}{cccccc}
31 & RS & RA & RB & 27 & Rc \\
0 & 6 & 11 & 16 & 21 & 31 \\
\end{array}
\]

\[
\begin{align*}
n & \leftarrow (RB)_{58:63} \\
r & \leftarrow \text{ROTL}_{64}(RS, n) \\
\text{if } (RB)_{57} = 0 & \text{ then } \\
\quad m & \leftarrow \text{MASK}(0, 63-n) \\
\text{else } m & \leftarrow 640 \\
RA & \leftarrow r \& m
\end{align*}
\]

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 Doubleword

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Source</th>
<th>Destination</th>
<th>X-form</th>
</tr>
</thead>
<tbody>
<tr>
<td>srd</td>
<td>RA,RS,RB</td>
<td>RA</td>
<td>(Rc=0)</td>
</tr>
<tr>
<td>srd.</td>
<td>RA,RS,RB</td>
<td>RA</td>
<td>(Rc=1)</td>
</tr>
</tbody>
</table>

\[
\begin{array}{cccccc}
31 & RS & RA & RB & 539 & Rc \\
0 & 6 & 11 & 16 & 21 & 31 \\
\end{array}
\]

\[
\begin{align*}
n & \leftarrow (RB)_{58:63} \\
r & \leftarrow \text{ROTL}_{64}(RS, 64-n) \\
\text{if } (RB)_{57} = 0 & \text{ then } \\
\quad m & \leftarrow \text{MASK}(n, 63) \\
\text{else } m & \leftarrow 640 \\
RA & \leftarrow r \& m
\end{align*}
\]

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. Zeros are supplied to the vacated positions on the left. 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**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>srad RA,RS,SH</td>
<td>(Rc=0)</td>
</tr>
<tr>
<td>srad. RA,RS,SH</td>
<td>(Rc=1)</td>
</tr>
</tbody>
</table>

\[
\begin{array}{cccccc}
\text{n} & \text{RA} & \text{RS} & \text{SH} & \text{RA} \\
0 & 31 & 6 & 11 & 16 & 21 & 413 & 30 & 31 \\
\end{array}
\]

\[
n \leftarrow \text{sh}_5 \ || \ \text{sh}_{0:4} \\
r \leftarrow \text{ROTL}_{64}(\text{RS}), 64-n \\
m \leftarrow \text{MASK}(n, 63) \\
s \leftarrow (\text{RS})_0 \\
\text{RA} \leftarrow \text{RAM} \ || \ ((^64s) \ & \ ^{64}m) \\
\text{carry} \leftarrow s \ & ((^64s) \ & \ ^{64}m) = 0 \]

The contents of register RS are shifted right SH bits. 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 (RS) is negative 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 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**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>extswsli RA,RS,SH</td>
<td>(Rc=0)</td>
</tr>
<tr>
<td>extswsli. RA,RS,SH</td>
<td>(Rc=1)</td>
</tr>
</tbody>
</table>

\[
\begin{array}{cccccc}
\text{n} & \text{RA} & \text{RS} & \text{SH} & \text{RA} \\
0 & 31 & 6 & 11 & 16 & 21 & 445 & 0 & 31 \\
\end{array}
\]

\[
n \leftarrow \text{sh}_5 \ || \ \text{sh}_{1:4} \\
r \leftarrow \text{ROTL}_{64}(\text{RA}), 64-n \\
m \leftarrow \text{MASK}(0, 63-n) \\
\text{RA} \leftarrow r \ & \ ^{64}m \\
\]

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 (cbcdtd and addg6s) and Decimal Floating-Point operands (cdt-bcd) See Chapter 5. for additional information.

**Convert Declets To Binary Coded Decimal X-form**

cdtbcd RA, RS

<table>
<thead>
<tr>
<th>31</th>
<th>RS</th>
<th>RA</th>
<th>///</th>
<th>282</th>
</tr>
</thead>
</table>

```
  do i = 0 to 1
  n ← i x 32
  RA_n:0:n+7 ← 0
  RA_n:8:n+19 ← DPD_TO_BCD( (RS)_n+12:n+21 )
  RA_n:20:n+31 ← DPD_TO_BCD( (RS)_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**

cbcdtd RA, RS

<table>
<thead>
<tr>
<th>31</th>
<th>RS</th>
<th>RA</th>
<th>///</th>
<th>314</th>
</tr>
</thead>
</table>

```
  do i = 0 to 1
  n ← i x 32
  RA_n:0:n+7 ← 0
  RA_n:8:n+19 ← BCD_TO_DPD( (RS)_n+12:n+21 )
  RA_n:20:n+31 ← BCD_TO_DPD( (RS)_n+22: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.

**Special Registers Altered:**
None

**Add and Generate Sixes XO-form**

addg6s RT,RA,RB

<table>
<thead>
<tr>
<th>31</th>
<th>RT</th>
<th>RA</th>
<th>RB</th>
<th>///</th>
<th>74</th>
</tr>
</thead>
</table>

```
  do i = 0 to 15
  dc_i ← carry_out( RA_{4xi:63} + RB_{4xi:63} )
  c ← \{ dc_0 \} || \{ dc_1 \} || \ldots || \{ dc_{15} \}
  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 4xn).

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.

```
  add r1,RA,r0
  add r2,r1,RB
  addg6s RT,r1,RB
  subf RT,RT,r3# 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.)

```
  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).
Move To/From Vector-Scalar Register Instructions

3.3.16 Move To/From Vector-Scalar Register Instructions

**Move From VSR Doubleword X-form**

\[ mfsvrd \ RA, XS \]

Let \( XS \) be the value \( 32 \times SX + S \).

The contents of doubleword element 0 of \( VSR[XS] \) are placed into \( GPR[RA] \).

For \( SX=0 \), \( mfsvrd \) is treated as a Floating-Point instruction in terms of resource availability.

For \( SX=1 \), \( mfsvrd \) is treated as a Vector instruction in terms of resource availability.

**Extended Mnemonics**

<table>
<thead>
<tr>
<th>( mfpdr ) RA, FRS</th>
<th>( mvrd ) RA, VRS</th>
</tr>
</thead>
<tbody>
<tr>
<td>( mfvrd ) RA, VRS</td>
<td>( mfvsrd ) RA, VRS+32</td>
</tr>
</tbody>
</table>

**Special Registers Altered**

None

**Data Layout for mfsvrd**

\( \text{src} = VSR[XS] \)

\[ .dword[0] \text{ unused} \]

\( \text{tgt} = GPR[RA] \)

\[ 0 \quad 64 \quad 127 \]

**Move From VSR Lower Doubleword X-form**

\[ mfsvrld \ RA, XS \]

Let \( XS \) be the value \( 32 \times SX + S \).

The contents of doubleword 1 of \( VSR[XS] \) are placed into \( GPR[RA] \).

For \( SX=0 \), \( mfsvrld \) is treated as a VSX instruction in terms of resource availability.

For \( SX=1 \), \( mfsvrld \) is treated as a Vector instruction in terms of resource availability.

**Special Registers Altered:**

None

**Data Layout for mfsvrld**

\( \text{src} = VSR[XS] \)

\[ \text{unused} \quad .dword[1] \]

\( \text{tgt} = GPR[RA] \)

\[ 0 \quad 64 \quad 127 \]
**Move From VSR Word and Zero X-form**

mfvsrwz RA,XS

<table>
<thead>
<tr>
<th>31</th>
<th>S</th>
<th>RA</th>
<th>///</th>
<th>115</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>16</td>
<td>11</td>
<td>21</td>
</tr>
</tbody>
</table>

Let $XS$ be the value $32 \times SX + S$.

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 Equivalent To**

- `mfprwz RA,FRS` → `mfvsrwz RA,FRS`
- `mfvrwz RA,VRS` → `mfvsrwz RA,VRS+32`

**Special Registers Altered**

- None

**Data Layout for mfvsrwz**

- **src = VSR[XS]**
  - unused
  - unused

- **tgt = GPR[RA]**
  - 0
  - 32
  - 64
  - 127
Move To VSR Doubleword X-form

\[ \text{mtvsrd} \text{ XT,RA} \]

<table>
<thead>
<tr>
<th>T</th>
<th>RA</th>
<th>0</th>
<th>179</th>
<th>XT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>15</td>
<td>21</td>
</tr>
</tbody>
</table>

if \( TX=0 \) & MSR.FP=0 then FP.Unavailable()
if \( TX=1 \) & MSR.VEC=0 then Vector.Unavailable()

\[ \text{VSR}[32\times TX+T].dword[0] \leftarrow \text{GPR}[RA] \]
\[ \text{VSR}[32\times TX+T].dword[1] \leftarrow 0xUUUU_UUUU_UUUU_UUUU \]

Let \( XT \) be the value \( 32 \times TX + T \).

The contents of \( \text{GPR}[RA] \) are placed into doubleword element 0 of \( \text{VSR}[XT] \).

The contents of doubleword element 1 of \( \text{VSR}[XT] \) are undefined.

For \( TX=0 \), \text{mtvsrd} is treated as a \text{Floating-Point} instruction in terms of resource availability.

For \( TX=1 \), \text{mtvsrd} is treated as a \text{Vector} instruction in terms of resource availability.

**Extended Mnemonics**

mtfprd FRT,RA
mtvrd VRT,RA

**Equivalent To**

mtvsrd FRT, RA
mtvsrd VRT+32, RA

**Special Registers Altered**

None

**Data Layout for mtvsrd**

src = GPR[RA]

tgt = VSR[XT]

[dword[0] undefined]

0 64 127

Move To VSR Word Algebraic X-form

\[ \text{mtvsrwa} \text{ XT,RA} \]

<table>
<thead>
<tr>
<th>T</th>
<th>RA</th>
<th>0</th>
<th>211</th>
<th>XT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>15</td>
<td>21</td>
</tr>
</tbody>
</table>

if \( TX=0 \) & MSR.FP=0 then FP.Unavailable()
if \( TX=1 \) & MSR.VEC=0 then Vector.Unavailable()

\[ \text{VSR}[32\times TX+T].dword[0] \leftarrow \text{EXTS64}[\text{GPR}[RA].bit[32:63]] \]
\[ \text{VSR}[32\times TX+T].dword[1] \leftarrow 0xUUUU_UUUU_UUUU_UUUU \]

Let \( XT \) be the value \( 32 \times TX + T \).

The two's-complement integer in bits 32:63 of \( \text{GPR}[RA] \) is sign-extended to 64 bits and placed into doubleword element 0 of \( \text{VSR}[XT] \).

The contents of doubleword element 1 of \( \text{VSR}[XT] \) are undefined.

For \( TX=0 \), \text{mtvsrwa} is treated as a \text{Floating-Point} instruction in terms of resource availability.

For \( TX=1 \), \text{mtvsrwa} is treated as a \text{Vector} instruction in terms of resource availability.

**Extended Mnemonics**

mtfprwa FRT,RA
mtrwa VRT,RA

**Equivalent To**

mtvsrwa FRT, RA
mtvsrwa VRT+32, RA

**Special Registers Altered**

None

**Data Layout for mtvsrwa**

src = GPR[RA]

undefined
tgt = VSR[XT]

[dword[0] undefined]

0 32 64 127
**Move To VSR Word and Zero X-form**

mtvsrwz  
XT,RA

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, mtvsrwz is treated as a Floating-Point instruction in terms of resource availability.

For TX=1, mtvsrwz is treated as a Vector instruction in terms of resource availability.

**Extended Mnemonics Equivalent To**

mtfprwz  FRT, RA
mtvwrz  VRT, RA

**Data Layout for mtvsrwz**

<table>
<thead>
<tr>
<th>src = GPR[RA]</th>
</tr>
</thead>
<tbody>
<tr>
<td>unused</td>
</tr>
<tr>
<td>tgt = VSR[XT]</td>
</tr>
<tr>
<td>.dword[0]</td>
</tr>
</tbody>
</table>

**Move To VSR Double Doubleword X-form**

mtvsrdd  
XT,RA,RB

Let XT be the value $32 \times TX + T$.

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**

<table>
<thead>
<tr>
<th>src = GPR[RA]</th>
</tr>
</thead>
<tbody>
<tr>
<td>src = GPR[RB]</td>
</tr>
<tr>
<td>tgt = VSR[XT]</td>
</tr>
<tr>
<td>.dword[0]</td>
</tr>
</tbody>
</table>
**Move To VSR Word & Splat X-form**

```assembly
mtvsrws XT,RA
```

Let $\text{XT}$ be the value $32 \times TX + T$.

The contents of bits 32:63 of \text{GPR}[\text{RA}] are placed into each word element of \text{VSR}[\text{XT}].

For $TX=0$, \textit{mtvsrws} is treated as a VSX instruction in terms of resource availability.

For $TX=1$, \textit{mtvsrws} is treated as a Vector instruction in terms of resource availability.

**Special Registers Altered:**

None
3.3.17 Move To/From System Register Instructions

The Move To Condition Register Fields instruction has a preferred form; see Section 1.9.1, “Preferred Instruction Forms” on page 23. 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 *mtrcf* instruction for compatibility with old software (written for a version of the architecture that precedes 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 791 for additional extended mnemonics.

**Move To Special Purpose Register XFX-form**

```
mtspr SPR_RS
```

```
  spr RS spr 467 /
  0 6 11 21 31
```

```
n ← spr[5:9] || spr[0:4]
switch [n]
case(13): see Book III
  case(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 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 00001</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>
</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 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>0</sub> = 0, the illegal instruction error handler is invoked.
- If spr<sub>0</sub> = 1, the system privileged instruction error handler is invoked.
If an attempt is made to execute `mtspr` specifying a TM SPR in other than Non-transactional state, with the exception of TFHAR in suspended state, a TM Bad Thing type Program interrupt is generated.

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</th>
<th>Equivalent to:</th>
</tr>
</thead>
<tbody>
<tr>
<td>mttxr Rx</td>
<td>mtspr 1,Rx</td>
</tr>
<tr>
<td>mttr 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

\[
\text{mfspr RT,SPR}
\]

\[
\begin{array}{c|c|c|c|c|c}
31 & 30 & 29 & 29 & 28 & 1 \\hline
\text{RT} & \text{sp} & \text{spr} & 339 & 1 & 31
\end{array}
\]

\[n \leftarrow \text{spr}_{5:9} \mid \mid \text{spr}_{0:4}\]

\[
\text{switch (n)}
\]

\[
\text{case(129): see Book III}
\]

\[
\text{case(808, 809, 810, 811):}
\]

\[
\text{default:}
\]

\[
\text{if length(SPR(n)) = 64 then}
\]

\[
\text{RT} \leftarrow \text{SPR(n)}
\]

\[
\text{else}
\]

\[
\text{RT} \leftarrow 32_0 \mid \mid \text{SPR(n)}
\]

The SPR field denotes a Special Purpose Register, encoded as shown in the table below. If the SPR field contains 129, the instruction references the Transaction Failure Instruction Address Register (TFIAR) and the result is dependent on the privilege with which it is executed. See Book III. 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.

If execution of this instruction is attempted specifying an SPR number that is not shown above, one of the following occurs.

- If spr0 = 0, the illegal instruction error handler is invoked.
- If spr0 = 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:

<table>
<thead>
<tr>
<th>Extended</th>
<th>Equivalent to:</th>
</tr>
</thead>
<tbody>
<tr>
<td>mfxer Rx</td>
<td>mfspr Rx,1</td>
</tr>
<tr>
<td>mfr Rx</td>
<td>mfspr Rx,8</td>
</tr>
<tr>
<td>mfctr Rx</td>
<td>mfspr Rx,9</td>
</tr>
</tbody>
</table>

**Note**

See the Notes that appear with *mfspr*.
Move to CR from XER Extended X-form

\[ \text{mcrxrx BF} \]

<table>
<thead>
<tr>
<th>31</th>
<th>BF</th>
<th>24</th>
<th>16</th>
<th>8</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>9</td>
<td>11</td>
<td>16</td>
<td>31</td>
</tr>
</tbody>
</table>

\[ \text{CR}_{4:BF+32:4:BF+35} \leftarrow \text{XER}_{OV:OV32:CA:CA32} \]

The contents of the OV, OV32, CA, and CA32 are copied to Condition Register field BF.

Special Registers Altered:

- CR field BF
**Move To One Condition Register Field**

**XFX-form**

\[
\text{mtocrf FXM},RS
\]

![Table]

<table>
<thead>
<tr>
<th>31</th>
<th>0</th>
<th>6</th>
<th>11</th>
<th>12</th>
<th>144</th>
<th>/</th>
</tr>
</thead>
</table>

\[\text{count} \leftarrow 0\]
\[\text{do } i = 0 \text{ to } 7\]
\[\quad \text{if } \text{FXM}_i = 1 \text{ then}\]
\[\quad \quad \text{n} \leftarrow i\]
\[\quad \text{count} \leftarrow \text{count} + 1\]
\[\quad \text{if } \text{count} = 1 \text{ then}\]
\[\quad \quad \text{CR}_{4\times n+32:4\times n+35} \leftarrow (\text{RS})_{4\times n+32:4\times n+35}\]
\[\quad \text{else } \text{CR} \leftarrow \text{undefined}\]

If exactly one bit of the FXM field is set to 1, let \( n \) be the position of that bit in the field \((0 \leq n \leq 7)\). The contents of bits \(4\times n+32:4\times n+35\) of register RS are placed into CR field \( n \) (CR bits \(4\times n+32:4\times n+35\)). Otherwise, the contents of the Condition Register are undefined.

**Special Registers Altered:**

- CR field selected by FXM

---

**Move To Condition Register Fields**

**XFX-form**

\[
\text{mtcrf FXM},RS
\]

![Table]

<table>
<thead>
<tr>
<th>31</th>
<th>0</th>
<th>6</th>
<th>11</th>
<th>12</th>
<th>144</th>
<th>/</th>
</tr>
</thead>
</table>

\[\text{mask} \leftarrow 4(\text{FXM}_0) || 4(\text{FXM}_1) || \ldots || 4(\text{FXM}_7)\]
\[\text{CR} \leftarrow (\text{RS})_{32:63} \& \text{mask} || (\text{CR} \& \neg \text{mask})\]

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 \( \text{FXM}_i = 1 \) then CR field \( i \) (CR bits \( 4\times i+32:4\times i+35 \)) 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: \( \text{mtrc Rx} \)
- Equivalent to: \( \text{mtcrf 0xFF,Rx} \)
Move From One Condition Register Field  
XFX-form

\[ \text{mfocrf} \rightarrow \text{RT,FXM} \]

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th>19</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>31</td>
<td>RT</td>
<td>1</td>
<td>FXM</td>
<td></td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>11</td>
<td>12</td>
<td>/</td>
<td>20</td>
</tr>
</tbody>
</table>

\( RT \leftarrow \text{undefined} \)

\( \text{count} \leftarrow 0 \)

\( \text{do } i = 0 \text{ to } 7 \)

\( \text{if } FXM_i = 1 \text{ then} \)

\( n \leftarrow i \)

\( \text{count} \leftarrow \text{count} + 1 \)

\( \text{if count} = 1 \text{ then} \)

\( RT \leftarrow 64^\text{0} \)

\( RT_{4\times n+32:4\times n+35} \leftarrow CR_{4\times n+32:4\times n+35} \)

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 \( 4\times n+32:4\times n+35 \)) are placed into bits \( 4\times n+32:4\times n+35 \) 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:** \text{mfocrf} is not backward compatible with processors that comply with versions of the architecture that precede Version 3.0 B. 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.

---

Move From Condition Register  
XFX-form

\[ \text{mfc} \rightarrow \text{RT} \]

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th>19</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>31</td>
<td>RT</td>
<td>11</td>
<td>12</td>
<td>20</td>
</tr>
</tbody>
</table>

\( RT \leftarrow 32^0 || CR \)

The contents of the Condition Register are placed into \( RT_{32:63} \). \( RT_{0:31} \) are set to 0.

**Special Registers Altered:**

None

---

Set Boolean  
X-form

\[ \text{setb} \rightarrow \text{RT,BFA} \]

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>19</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>31</td>
<td>RT</td>
<td>11</td>
<td>12</td>
<td>19</td>
<td></td>
</tr>
</tbody>
</table>

\( \text{if } CR_{4\times BFA+32}=1 \text{ then} \)

\( RT \leftarrow 0xFFFF_FFFF_FFFF_FFFF \)

\( \text{else if } CR_{4\times BFA+33}=1 \text{ then} \)

\( RT \leftarrow 0000_0000_0000_0001 \)

\( \text{else} \)

\( RT \leftarrow 0000_0000_0000_0000 \)

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
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 (VXID)
  - Zero–Zero (VXZDZ)
  - Infinity–Zero (VXIMZ)
  - Invalid Compare (VXVC)
  - Software-Defined Condition (VXSOFT)
  - Invalid Square Root (VXSQRT)
Invalid Integer Convert (VXCVI)

- 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 4.2.2, “Floating-Point Status and Control Register” on page 124 for a description of these exception and enable bits, and Section 4.4, “Floating-Point Exceptions” on page 132 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 124.

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.

```
| FPR 0 |
| FPR 1 |
| ...  |
| ...  |
| FPR 30 |
| FPR 31 |
```

Figure 45. Floating-Point Registers

4.2.2 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 32:55 are status bits. Bits 56:63 are control bits.

The exception 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 mcrfs, mtfsi, mtfsf, or mtfsb0 instruction. The exception summary bits in the FPSCR (FX, FEX, and VX, which are bits 32:34) are not considered to be “exception bits”, and only FX is sticky.

FEX 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.

```
| FPSCR |
```

Figure 46. Floating-Point Status and Control Register

The bit definitions for the FPSCR are as follows.

<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>Floating-Point Exception Summary (FX) Every floating-point instruction, except mtfsi and mtfsf, implicitly sets FPSCR_EX to 1 if that instruction causes any of the floating-point exception bits in the FPSCR to change from 0 to 1. mcrfs, mtfsi, mtfsf, mtfsb0, and mtfsb1 can alter FPSCR_EX explicitly.</td>
</tr>
</tbody>
</table>
Floating-Point Facility

Chapter 4

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 FPSCR_FEX explicitly.

Floating-Point Invalid Operation Exception Summary (VX)
This bit is the OR of all the Invalid Operation exception bits. `mcrfs`, `mtfsfi`, `mtfsf`, `mtfsb0`, and `mtfsb1` cannot alter FPSCR_VX explicitly.

Floating-Point Overflow Exception (OX)
See Section 4.4.3, “Overflow Exception” on page 135.

Floating-Point Underflow Exception (UX)
See Section 4.4.4, “Underflow Exception” on page 136.

Floating-Point Zero Divide Exception (ZX)
See Section 4.4.2, “Zero Divide Exception” on page 134.

Floating-Point Inexact Exception (XX)
See Section 4.4.5, “Inexact Exception” on page 136.

FPSCR_FXX is a sticky version of FPSCR_FI (see below). Thus the following rules completely describe how FPSCR_FXX is set by a given instruction.

- If the instruction affects FPSCR_FI, the new value of FPSCR_FXX is obtained by ORing the old value of FPSCR_FXX with the new value of FPSCR_FI.
- If the instruction does not affect FPSCR_FI, the value of FPSCR_FXX is unchanged.

Floating-Point Invalid Operation Exception (SNaN) (VXSNAN)
See Section 4.4.1, “Invalid Operation Exception” on page 134.

Floating-Point Invalid Operation Exception (0 / 0) (VXZDZ)
See Section 4.4.1.

Floating-Point Invalid Operation Exception (∞ / 0) (VXIMZ)
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 131. 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 FPSCR_FXX, above, regarding the relationship between FPSCR_FI and FPSCR_FXX.

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.

Programming Note
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 127.

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 127. 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`, `mtfsf`, `mtfsb0`, or `mtfsb1`. See Section 4.4.1.

**Programming Note**

FPSCR[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 135.

Floating-Point Underflow Exception Enable (UE)

See Section 4.4.4, “Underflow Exception” on page 136.

Floating-Point Zero Divide Exception Enable (ZE)

See Section 4.4.2, “Zero Divide Exception” on page 134.

Floating-Point Inexact Exception Enable (XE)

See Section 4.4.5, “Inexact Exception” on page 136.

**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[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.
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.

<table>
<thead>
<tr>
<th>Format</th>
<th>Single</th>
<th>Double</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exponent Bias</td>
<td>+127</td>
<td>+1023</td>
</tr>
<tr>
<td>Maximum Exponent</td>
<td>+127</td>
<td>+1023</td>
</tr>
<tr>
<td>Minimum Exponent</td>
<td>−126</td>
<td>−1022</td>
</tr>
</tbody>
</table>

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.

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.
**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** (± 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:

\[ \text{DEN} = (-1)^s \times 2^{\text{Emin}} \times (0.\text{fraction}) \]

where \( \text{Emin} \) 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:

\[ -\infty < \text{every finite number} < +\infty \]

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 134.

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 (FPSCRVE=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 (FRA) is a NaN
  then FRT \( \leftarrow \) (FRA)
  else if (FRB) is a NaN
    then if instruction is frsp
      then FRT \( \leftarrow \) (FRB)0:34 \( \parallel \) 290
      else FRT \( \leftarrow \) (FRB)
    else if (FRC) is a NaN
      then FRT \( \leftarrow \) (FRC)
      else if generated QNaN
        then FRT \( \leftarrow \) generated QNaN

If the operand specified by FRA is a NaN, then that NaN is stored as the result. Otherwise, if the operand specified by FRB is a NaN (if the instruction specifies an 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 frsp. Otherwise, if the operand specified by FRC is a NaN (if the instruction specifies an 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
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 Reciprocals 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 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 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 4.5.1, “Execution Model for IEEE Operations” on page 137) 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 136) 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_{RN} and to round toward zero. These instructions may cause Invalid Operation (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 FR and 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 FR is set to 1, otherwise FR is set to 0. If the result is inexact then FI is set to 1, otherwise FI is set to zero. The Round to Integer instructions are exceptions to this rule, setting FR and FI to 0. The Estimate instructions set FR and FI to undefined values. The remaining floating-point instructions do not alter FR and 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 Z1 and Z2 bound Z as the next larger and next smaller numbers representable in the target format. Then Z1 or Z2 can be used to approximate the result in the target format.

Figure 52 shows the relation of Z, Z1, and Z2 in this case. The following rules specify the rounding in the four modes. “LSB” means “least significant bit”.

**Round to Nearest**

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 0).

**Round toward Zero**

Choose the smaller in magnitude (Z1 or Z2).

**Round toward +Infinity**

Choose Z1.

**Round toward -Infinity**

Choose Z2.

See Section 4.5.1, “Execution Model for IEEE Operations” on page 137 for a detailed explanation of rounding.
4.4 Floating-Point Exceptions

This architecture defines the following floating-point exceptions:

- Invalid Operation Exception
  - SNaN
    - Infinity-Infinity
    - Infinity-Infinity
    - Zero-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 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 FPSCR[XSOF] 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 133), 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 mtsfi or mtlsf, 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 (\(\times 0\)) 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.

### FE0 FE1 Description

<table>
<thead>
<tr>
<th>0 0</th>
<th>Ignore Exceptions Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Floating-point exceptions do not cause the system floating-point enabled exception error handler to be invoked.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0 1</th>
<th>Imprecise Nonrecoverable Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>The system floating-point enabled exception error handler is invoked at some point at or beyond 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>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1 0</th>
<th>Imprecise Recoverable Mode</th>
</tr>
</thead>
<tbody>
<tr>
<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 so 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>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1 1</th>
<th>Precise Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>The system floating-point enabled exception error handler is invoked precisely at the instruction that caused the enabled exception.</td>
<td></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 ($\infty \div \infty$)
- Division of zero by zero ($0 \div 0$)
- Multiplication of infinity by zero ($\infty \times 0$)
- Ordered comparison involving a NaN (Invalid Compare)
- Square root or reciprocal square root of a negative (and nonzero) 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 `mtfssb` instruction is executed that sets FPSCR 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 (FPSCRVE=1) and an Invalid Operation Exception occurs, the following actions are taken:

1. One or two Invalid Operation Exceptions are set
   - FPSCRVXSNAN (if SNaN)
   - FPSCRVXISI (if $\infty - \infty$)
   - FPSCRVXIDI (if $\infty + \infty$)
   - FPSCRVXZDZ (if $0 \div 0$)
   - FPSCRVXIMZ (if $\infty \times 0$)
   - FPSCRVXVC (if invalid comp)
   - FPSCRVXSQRT (if invalid sqrt)
   - FPSCRVXCVI (if invalid int cvrt)

2. If the operation is an arithmetic or Floating Round to Single-Precision operation,
   - the target FPR is set to a Quiet NaN
   - FPSCRFR, FI are set to zero
   - FPSCRFPCC 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 $+\infty$, and to the most negative 64-bit integer if the operand in FRB is a negative number, $-\infty$, or NaN
     - FPSCRFR, FI are set to zero
     - FPSCRFPCC is undefined

4. If the operation is a convert to 32-bit integer operation,
   - the target FPR is set as follows:
     - FRT0:31 are undefined
     - FRT32:63 are set to the most positive 32-bit integer if the operand in FRB is a positive number or $+\infty$, and to the most negative 32-bit integer if the operand in FRB is a negative number, $-\infty$, or NaN
     - FPSCRFR, FI are set to zero
     - FPSCRFPCC is undefined

5. If the operation is a compare,
   - FPSCRFR, FI are unchanged
   - FPSCRFPCC is set to reflect unordered

6. If an `mtfsfi`, `mtfsf`, or `mtfssb` instruction is executed that sets FPSCR to 1,
   - The FPSCR is set as specified in the instruction description.

When Invalid Operation Exception is disabled (FPSCRVE=0) and an Invalid Operation Exception occurs, the following actions are taken:

1. One or two Invalid Operation Exceptions are set
   - FPSCRVXSNAN (if SNaN)
   - FPSCRVXISI (if $\infty - \infty$)
   - FPSCRVXIDI (if $\infty + \infty$)
   - FPSCRVXZDZ (if $0 \div 0$)
   - FPSCRVXIMZ (if $\infty \times 0$)
   - FPSCRVXVC (if invalid comp)
   - FPSCRVXSQRT (if invalid sqrt)
   - FPSCRVXCVI (if invalid int cvrt)

2. If the operation is an arithmetic or Floating Round to Single-Precision operation,
   - the target FPR is set to a Quiet NaN
   - FPSCRFR, FI are set to zero
   - FPSCRFPCC 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 $+\infty$, and to the most negative 64-bit integer if the operand in FRB is a negative number, $-\infty$, or NaN
     - FPSCRFR, FI are set to zero
     - FPSCRFPCC is undefined

4. If the operation is a convert to 32-bit integer operation,
   - the target FPR is set as follows:
     - FRT0:31 are undefined
     - FRT32:63 are set to the most positive 32-bit integer if the operand in FRB is a positive number or $+\infty$, and to the most negative 32-bit integer if the operand in FRB is a negative number, $-\infty$, or NaN
     - FPSCRFR, FI are set to zero
     - FPSCRFPCC is undefined

5. If the operation is a compare,
   - FPSCRFR, FI are unchanged
   - FPSCRFPCC is set to reflect unordered

6. If an `mtfsfi`, `mtfsf`, or `mtfssb` instruction is executed that sets FPSCR 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 or frsqrte) 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 (FPSCRZE=1) and a Zero Divide Exception occurs, the following actions are taken:

1. Zero Divide Exception is set
   \[ \text{FPSCR}_{ZE} \leftarrow 1 \]
2. The target FPR is unchanged
3. FPSCRFRFI are set to zero
4. FPSCRFPREF is unchanged

When Zero Divide Exception is disabled (FPSCRZE=0) and a Zero Divide Exception occurs, the following actions are taken:

1. Zero Divide Exception is set
   \[ \text{FPSCR}_{ZE} \leftarrow 1 \]
2. The target FPR is set to \( \pm \infty \), where the sign is determined by the XOR of the signs of the operands
3. FPSCRFRFI are set to zero
4. FPSCRFPREF is set to indicate the class and sign of the result (\( \pm \infty \))

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 (FPSCRXE=1) and an Overflow Exception occurs, the following actions are taken:

1. Overflow Exception is set
   \[ \text{FPSCR}_{OX} \leftarrow 1 \]
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. FPSCRFPREF is set to indicate the class and sign of the result (\( \pm \infty \) or \( \pm \text{Normal Number} \))

When Overflow Exception is disabled (FPSCRXE=0) and an Overflow Exception occurs, the following actions are taken:

1. Overflow Exception is set
   \[ \text{FPSCR}_{OX} \leftarrow 1 \]
2. Inexact Exception is set
   \[ \text{FPSCR}_{XX} \leftarrow 1 \]
3. The result is determined by the rounding mode (FPSCRRN) and the sign of the intermediate result as follows:
   - Round to Nearest
     Store \( \pm \infty \), 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 \( +\infty \)
     For negative overflow, store \( -\infty \); for positive overflow, store \( +\infty \)
   - Round toward \(-\infty \)
     For negative overflow, store \( -\infty \); for positive overflow, store the format’s largest finite number
4. The result is placed into the target FPR
5. FPSCRFR is undefined
6. FPSCRFI is set to 1
7. FPSCRFPREF is set to indicate the class and sign of the result (\( \pm \infty \) or \( \pm \text{Normal Number} \))
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 (FPSCR_{UE}=0) then the intermediate result is denormalized (see Section 4.3.4, “Normalization and Denormalization” on page 129) and rounded (see Section 4.3.6, “Rounding” on page 131) 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 (FPSCR_{UE}=1) and an Underflow Exception occurs, the following actions are taken:

1. Underflow Exception is set
   FPSCR_{UX} ← 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. FPSCR_{FRF} is set to indicate the class and sign of the result (± Normalized Number)

When Underflow Exception is disabled (FPSCR_{UE}=0) and an Underflow Exception occurs, the following actions are taken:

1. Underflow Exception is set
   FPSCR_{UX} ← 1
2. The rounded result is placed into the target FPR
3. FPSCR_{FRF} 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
   FPSCR_{XX} ← 1
2. The rounded or overflowed result is placed into the target FPR
3. FPSCR_{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 **fcfid** 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 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 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.

![Figure 53. IEEE 64-bit execution model](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.

The 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 R bit, due either to 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. 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).

![Figure 54. Interpretation of G, R, and X bits](image)

<table>
<thead>
<tr>
<th>G R X</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0</td>
<td>IR is exact</td>
</tr>
<tr>
<td>0 0 1</td>
<td>IR closer to NL</td>
</tr>
<tr>
<td>0 1 0</td>
<td>IR midway between NL and NH</td>
</tr>
<tr>
<td>0 1 1</td>
<td>IR closer to NH</td>
</tr>
<tr>
<td>1 0 0</td>
<td>IR midway between NL and NH</td>
</tr>
<tr>
<td>1 0 1</td>
<td>IR closer to NH</td>
</tr>
<tr>
<td>1 1 0</td>
<td>IR closer to NH</td>
</tr>
<tr>
<td>1 1 1</td>
<td>IR closer to NH</td>
</tr>
</tbody>
</table>

![Figure 55. Location of the Guard, Round, and Sticky bits in the IEEE execution model](image)

**Format** | **Guard** | **Round** | **Sticky** |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Double</td>
<td>G bit</td>
<td>R bit</td>
<td>X bit</td>
</tr>
</tbody>
</table>

Figure 55 shows the positions of the Guard, Round, and Sticky bits for double-precision and single-precision floating-point numbers relative to the accumulator illustrated in Figure 53.
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 FPSCR_RN as described in Section 4.3.6, “Rounding” on page 131. Using Z1 and Z2 as defined on page 131, 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 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.

![Figure 56. Multiply-add 64-bit execution model](image)

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 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 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.

![Figure 57. Location of the Guard, Round, and Sticky bits in the multiply-add execution model](image)

The rules for rounding the intermediate result are the same as those given in Section 4.5.1.
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 1.11.3, “Effective Address Calculation” on page 27.

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 Mne
omics” on page 802.

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 143. 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 WORD0:31 be the floating-point single-precision operand accessed from storage.

Load Floating-Point Single D-form

<table>
<thead>
<tr>
<th>Ifs</th>
<th>FRT,D(RA)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>48</td>
</tr>
<tr>
<td>6</td>
<td>FRT</td>
</tr>
<tr>
<td>11</td>
<td>RA</td>
</tr>
<tr>
<td>16</td>
<td>D</td>
</tr>
</tbody>
</table>

if RA = 0 then b ← 0
else b ← [RA]
EA ← b + EXTS(D)
FRT ← DOUBLE(MEM(EA, 4))

Let the effective address (EA) be the sum (RA[0]+D).

Normal Operand

if WORD1:8 > 0 and WORD1:8 < 255 then
FRT0:1 ← WORD0:1
FRT2 ← ¬WORD1
FRT3 ← ¬WORD1
FRT4 ← ¬WORD1
FRT5:63 ← WORD2:31 | 290

Denormalized Operand

if WORD1:8 = 0 and WORD9:31 ≠ 0 then
sign ← WORD0
exp ← -126
frac0:52 ← 0b0 || WORD9:31 | 290
normalize the operand
do while frac0 = 0
frac0:52 ← frac1:52 || 0b0
exp ← exp - 1
FRT0 ← sign
FRT1:11 ← exp + 1023
FRT12:63 ← frac1:52

Zero / Infinity / NaN

if WORD1:8 = 255 or WORD1:31 = 0 then
FRT0:1 ← WORD0:1
FRT2 ← WORD1
FRT3 ← WORD1
FRT4 ← WORD1
FRT5:63 ← WORD2:31 | 290

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 Indexed X-form**

\[ \text{Ifsx} \quad \text{FRT}, \text{RA}, \text{RB} \]

<table>
<thead>
<tr>
<th>31</th>
<th>FRT</th>
<th>RA</th>
<th>RB</th>
<th>535</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

If RA = 0 then b \( \leftarrow 0 \)
else \( b \leftarrow (\text{RA}) \)
EA \( \leftarrow b + (\text{RB}) \)
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 140) and placed into register FRT.

**Special Registers Altered:** None

**Load Floating-Point Single with Update D-form**

\[ \text{Ifsu} \quad \text{FRT}, \text{D}(\text{RA}) \]

<table>
<thead>
<tr>
<th>49</th>
<th>FRT</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 \( \leftarrow (\text{RA}) + \text{EXTS(D)} \)
FRT \( \leftarrow \text{DOUBLE(MEM(EA, 4))} \)
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 140) 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

Ifsxu FRT,RA,RB

<table>
<thead>
<tr>
<th>FRT</th>
<th>RA</th>
<th>RB</th>
<th>567</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>6</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

EA ← (RA) + (RB)
FRT ← DOUBLE(MEM(EA, 4))
RA ← EA

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 140) 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>FRT</th>
<th>RA</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

if RA = 0 then b ← 0
else
b ← (RA)
EA ← b + EXTS(D)
FRT ← MEM(EA, 8)

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

Special Registers Altered:
None

Load Floating-Point Double Indexed X-form

Ifdx FRT,RA,RB

<table>
<thead>
<tr>
<th>FRT</th>
<th>RA</th>
<th>RB</th>
<th>599</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>6</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</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)+(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>FRT</th>
<th>RA</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>51</td>
<td>6</td>
<td></td>
</tr>
</tbody>
</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

**Instruction:** `lfdux FRT,RA,RB

```
<table>
<thead>
<tr>
<th>31</th>
<th>8</th>
<th>16</th>
<th>21</th>
<th>631</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>
```

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

**Instruction:** `lfiwax FRT,RA,RB

```
<table>
<thead>
<tr>
<th>31</th>
<th>8</th>
<th>16</th>
<th>21</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 ← 0
else b ← (RA)
EA ← b + (RB)
FRT ← EXT(MEM(EA, 4))

Let the effective address (EA) be the sum (RA)+(RB).

The word in storage addressed by EA is loaded into FRT{32:63}. 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 and Zero Indexed X-form

**Instruction:** `lfiwzx FRT,RA,RB

```
<table>
<thead>
<tr>
<th>31</th>
<th>8</th>
<th>16</th>
<th>21</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 ← 0
else b ← (RA)
EA ← b + (RB)
FRT ← EXT(MEM(EA, 4))

Let the effective address (EA) be the sum (RA)+0+(RB).

The word in storage addressed by EA is loaded into FRT_{32:63}. 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 147. 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)**

if \( \text{FRS}_{1:11} \geq 896 \) or \( \text{FRS}_{1:63} = 0 \) then

\[
\begin{align*}
\text{WORD}_{0:1} & \leftarrow \text{FRS}_{0:1} \\
\text{WORD}_{2:31} & \leftarrow \text{FRS}_{5:34}
\end{align*}
\]

**Denormalization Required**

if \( 874 \leq \text{FRS}_{1:11} \leq 896 \) then

\[
\begin{align*}
\text{sign} & \leftarrow \text{FRS}_0 \\
\text{exp} & \leftarrow \text{FRS}_{1:11} - 1023 \\
\text{frac}_{0:52} & \leftarrow 0b1 \| \text{FRS}_{12:63} \\
\text{denormalize operand} \\
\text{do while } \text{exp} < -126 \\
\text{frac}_{0:52} & \leftarrow 0b0 \| \text{frac}_{0:51} \\
\text{exp} & \leftarrow \text{exp} + 1 \\
\text{WORD}_0 & \leftarrow \text{sign} \\
\text{WORD}_{1:8} & \leftarrow \text{0000} \\
\text{WORD}_{9:31} & \leftarrow \text{frac}_{1:23}
\end{align*}
\]

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 \( \text{WORD} \) 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 \( \text{WORD} \) 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 \( \text{RA} \neq 0 \), 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

\textbf{stfs} \ FRS,D(RA)

\begin{tabular}{|c|c|c|c|}
\hline
52 & FRS & RA & D \\
\hline
0 & 6 & 11 & 16 & 31 \\
\hline
\end{tabular}

if RA = 0 then b $\leftarrow$ 0
else \quad b $\leftarrow$ (RA)
EA $\leftarrow$ b + EXTS(D)
MEM(EA, 4) $\leftarrow$ SINGLE((FRS))

Let the effective address (EA) be the sum (RA|0)+D.

The contents of register FRS are converted to single format (see page 144) and stored into the word in storage addressed by EA.

Special Registers Altered:
None

Store Floating-Point Single Indexed X-form

\textbf{stfsx} \ FRS,RA,RB

\begin{tabular}{|c|c|c|c|c|}
\hline
31 & FRS & RA & RB & 663 \\
\hline
0 & 6 & 11 & 16 & 21 & 31 \\
\hline
\end{tabular}

if RA = 0 then b $\leftarrow$ 0
else \quad b $\leftarrow$ (RA)
EA $\leftarrow$ b + (RB)
MEM(EA, 4) $\leftarrow$ SINGLE((FRS))

Let the effective address (EA) be the sum (RA|0)+(RB).

The contents of register FRS are converted to single format (see page 144) and stored into the word in storage addressed by EA.

Special Registers Altered:
None

Store Floating-Point Single with Update D-form

\textbf{stfsu} \ FRS,D(RA)

\begin{tabular}{|c|c|c|c|}
\hline
53 & FRS & RA & D \\
\hline
0 & 6 & 11 & 16 & 31 \\
\hline
\end{tabular}

EA $\leftarrow$ (RA) + EXTS(D)
MEM(EA, 4) $\leftarrow$ SINGLE((FRS))
RA $\leftarrow$ EA

Let the effective address (EA) be the sum (RA|0)+D.

The contents of register FRS are converted to single format (see page 144) 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

\textbf{stfsux} \ FRS,RA,RB

\begin{tabular}{|c|c|c|c|c|}
\hline
31 & FRS & RA & RB & 695 \\
\hline
0 & 6 & 11 & 16 & 21 & 31 \\
\hline
\end{tabular}

EA $\leftarrow$ (RA) + (RB)
MEM(EA, 4) $\leftarrow$ 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 144) 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**

```
stfd FRS,D(RA)
```

<table>
<thead>
<tr>
<th></th>
<th>FRS</th>
<th>RA</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>54</td>
<td>6</td>
<td>16</td>
</tr>
</tbody>
</table>

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

\( EA \leftarrow b + \text{EXTS}(D) \)
\( \text{MEM}(EA, 8) \leftarrow (FRS) \)

Let the effective address (EA) be the sum (RA(0)+D).
The contents of register FRS are stored into the doubleword in storage addressed by EA.

**Special Registers Altered:**

None

**Store Floating-Point Double Indexed X-form**

```
stfdx FRS,RA,RB
```

<table>
<thead>
<tr>
<th></th>
<th>FRS</th>
<th>RA</th>
<th>RB</th>
<th>727</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>31</td>
<td>6</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

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

\( EA \leftarrow b + (RB) \)
\( \text{MEM}(EA, 8) \leftarrow (FRS) \)

Let the effective address (EA) be the sum (RA(0)+RB).
The contents of register FRS are stored into the doubleword in storage addressed by EA.

**Special Registers Altered:**

None

**Store Floating-Point Double with Update D-form**

```
stfdu FRS,D(RA)
```

<table>
<thead>
<tr>
<th></th>
<th>FRS</th>
<th>RA</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>55</td>
<td>6</td>
<td>16</td>
</tr>
</tbody>
</table>

\( EA \leftarrow (RA) + \text{EXTS}(D) \)
\( \text{MEM}(EA, 8) \leftarrow (FRS) \)
\( RA \leftarrow EA \)

Let the effective address (EA) be the sum (RA)+D.
The contents of register FRS are 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 Floating-Point Double with Update Indexed X-form**

```
stfdux FRS,RA,RB
```

<table>
<thead>
<tr>
<th></th>
<th>FRS</th>
<th>RA</th>
<th>RB</th>
<th>759</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>31</td>
<td>6</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

\( EA \leftarrow (RA) + (RB) \)
\( \text{MEM}(EA, 8) \leftarrow (FRS) \)
\( RA \leftarrow EA \)

Let the effective address (EA) be the sum (RA)+(RB).
The contents of register FRS are 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 Floating-Point as Integer Word
Indexed X-form

\[
\text{stfiwx FRS,RA,RB}
\]

\[
\begin{array}{cccccc}
0 & 31 & \text{FRS} & \text{RA} & \text{RB} & 983 \\
6 & 11 & 16 & 21 & 31 & \\
\end{array}
\]

if RA = 0 then \( b \leftarrow 0 \)
else \( b \leftarrow (\text{RA}) \)
\( \text{EA} \leftarrow b + (\text{RB}) \)
\( \text{MEM}(\text{EA}, 4) \leftarrow (\text{FRS})_{32:63} \)

Let the effective address (EA) be the sum (RA|0)+(RB).

\( (\text{FRS})_{32:63} \) 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 \( \text{frsp} \), 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.

\begin{center}
\textbf{Programming Note}
\end{center}

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.
Load Floating-Point Double Pair DS-form

\[ lfdp \quad \text{FRTp,DS(RA)} \]

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>DS</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>57</td>
<td>FRTp</td>
<td>RA</td>
<td>DS</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

if RA = 0 then b ← 0
else b ← (RA)
EA ← b + EXTS(DS||0b00)
FRTpeven ← MEM(EA, 8)
FRTpodd ← MEM(EA+8, 8)

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 FRTp.
The doubleword in storage addressed by EA+8 is placed into the odd-numbered register of FRTp.
If FRTp is odd, the instruction form is invalid.

Special Registers Altered:
None

Load Floating-Point Double Pair Indexed X-form

\[ lfdpx \quad \text{FRTp,RA,RB} \]

<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>31</td>
<td>FRTp</td>
<td>RA</td>
<td>RB</td>
<td>791</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

if RA = 0 then b ← 0
else b ← (RA)
EA ← b + (RB)
FRTpeven ← MEM(EA, 8)
FRTpodd ← MEM(EA+8, 8)

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 FRTp.
The doubleword in storage addressed by EA+8 is placed into the odd-numbered register of FRTp.
If FRTp is odd, the instruction form is invalid.

Special Registers Altered:
None

Store Floating-Point Double Pair DS-form

\[ stfdp \quad \text{FRSp,DS(RA)} \]

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>DS</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>61</td>
<td>FRSp</td>
<td>RA</td>
<td>DS</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

if RA = 0 then b ← 0
else b ← (RA)
EA ← b + EXTS(DS||0b00)
MEM(EA, 8) ← FRSp_even
MEM(EA+8, 8) ← FRSp_odd

Let the effective address (EA) be the sum (RA|0) + (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

\[ stfdpx \quad \text{FRSp,RA,RB} \]

<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>31</td>
<td>FRSp</td>
<td>RA</td>
<td>RB</td>
<td>919</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

if RA = 0 then b ← 0
else b ← (RA)
EA ← b + (RB)
MEM(EA, 8) ← FRSp_even
MEM(EA+8, 8) ← FRSp_odd

Let the effective address (EA) be the sum (RA|0) + (RB).

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.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>FMT</th>
<th>63</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>264</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fmr</code> FRT,FRB</td>
<td></td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>72</td>
<td></td>
</tr>
<tr>
<td><code>fmr</code> FRT,FRB</td>
<td></td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>72</td>
<td></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>FMT</th>
<th>63</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>40</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fneg</code> FRT,FRB</td>
<td></td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>72</td>
<td></td>
</tr>
<tr>
<td><code>fneg</code> FRT,FRB</td>
<td></td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>72</td>
<td></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>FMT</th>
<th>63</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>264</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fabs</code> FRT,FRB</td>
<td></td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>72</td>
<td></td>
</tr>
<tr>
<td><code>fabs</code> FRT,FRB</td>
<td></td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>72</td>
<td></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 Copy Sign X-form

<table>
<thead>
<tr>
<th>Instruction</th>
<th>FMT</th>
<th>63</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>8</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fcpsgn</code> FRT,FRA,FRB</td>
<td></td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>72</td>
<td></td>
</tr>
<tr>
<td><code>fcpsgn</code> FRT,FRA,FRB</td>
<td></td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>72</td>
<td></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)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>FMT</th>
<th>63</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>136</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fnabs</code> FRT,FRB</td>
<td></td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>72</td>
<td></td>
</tr>
<tr>
<td><code>fnabs</code> FRT,FRB</td>
<td></td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>72</td>
<td></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 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 [Single] A-form**

- fadd  FRT,FRA,FRB  \((Rc=0)\)
- fadd. FRT,FRA,FRB  \((Rc=1)\)

```
   63  FRT  FRA  FRB  ///  21  20
   0   6   11  16  21  26  31
```

- fadds  FRT,FRA,FRB  \((Rc=0)\)
- fadds. FRT,FRA,FRB  \((Rc=1)\)

```
   59  FRT  FRA  FRB  ///  21  20
   0   6   11  16  21  26  31
```

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 the Floating-Point Rounding Control field RN of the FPSCR 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.

FPSCR\(_{FPRF}\) is set to the class and sign of the result, except for Invalid Operation Exceptions when FPSCR\(_{VE}\)=1.

**Special Registers Altered:**
- FPRF FR FI
- FX OX UX XX
- VXSNAN VXISI
- CR1  \(\text{if } Rc=1\)

**Floating Subtract [Single] A-form**

- fsub  FRT,FRA,FRB  \((Rc=0)\)
- fsub. FRT,FRA,FRB  \((Rc=1)\)

```
   63  FRT  FRA  FRB  ///  21  20
   0   6   11  16  21  26  31
```

- fsubs  FRT,FRA,FRB  \((Rc=0)\)
- fsubs. FRT,FRA,FRB  \((Rc=1)\)

```
   59  FRT  FRA  FRB  ///  21  20
   0   6   11  16  21  26  31
```

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 the Floating-Point Rounding Control field RN of the FPSCR 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.

FPSCR\(_{FPRF}\) is set to the class and sign of the result, except for Invalid Operation Exceptions when FPSCR\(_{VE}\)=1.

**Special Registers Altered:**
- FPRF FR FI
- FX OX UX XX
- VXSNAN VXISI
- CR1  \(\text{if } Rc=1\)
Floating Multiply [Single] A-form

\[ \text{fmul} \] FRT,FRA,FRC (Rc=0)
\[ \text{fmul} \] FRT,FRA,FRC (Rc=1)

The floating-point operand in register FRA is multiplied by the floating-point operand in register 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 the Floating-Point Rounding Control field RN of the FPSCR and placed into register FRT.

Floating-point multiplication is based on exponent addition and multiplication of the significands.

FPSCR\text{FPRF} is set to the class and sign of the result, except for Invalid Operation Exceptions when FPSCR\text{VE}=1.

\textbf{Special Registers Altered:}
- FPRF FR FI
- FX OX UX XX
- VXSNAN VXIMZ
- CR1 (if Rc=1)

Floating Divide [Single] A-form

\[ \text{fdiv} \] FRT,FRA,FRB (Rc=0)
\[ \text{fdiv} \] FRT,FRA,FRB (Rc=1)

The floating-point operand in register FRA is divided by the floating-point operand in register 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 the Floating-Point Rounding Control field RN of the FPSCR and placed into register FRT.

Floating-point division is based on exponent subtraction and division of the significands.

FPSCR\text{FPRF} is set to the class and sign of the result, except for Invalid Operation Exceptions when FPSCR\text{VE}=1 and Zero Divide Exceptions when FPSCR\text{ZE}=1.

\textbf{Special Registers Altered:}
- FPRF FR FI
- FX OX UX XX
- VXSNAN VXIDI VXZDZ
- CR1 (if Rc=1)
**Floating Square Root [Single] A-form**

\[ \text{fsqrt} \ FRT,FRB \quad \text{(Rc=0)} \]
\[ \text{fsqrt.} \ FRT,FRB \quad \text{(Rc=1)} \]

<table>
<thead>
<tr>
<th>FRC</th>
<th>FRT</th>
<th>FRB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>63</td>
<td>11</td>
</tr>
<tr>
<td>6</td>
<td>21</td>
<td>26</td>
</tr>
</tbody>
</table>

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 the Floating-Point Rounding Control field RN of the FPSCR 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>VXSQAN</td>
</tr>
<tr>
<td>QNaN</td>
<td>QNaN</td>
<td>None</td>
</tr>
</tbody>
</table>

\(^1\) No result if FPSCR\(_{VE}\) = 1

FPSCR\(_{PFRF}\) is set to the class and sign of the result, except for Invalid Operation Exceptions when FPSCR\(_{VE}\)=1.

**Special Registers Altered:**

FPRF FR FI FX OX UX XX
VXSQAN VXSQRT
CR1 (if Rc=1)

---

**Floating Reciprocal Estimate [Single] A-form**

\[ \text{fre} \ FRT,FRB \quad \text{(Rc=0)} \]
\[ \text{fre.} \ FRT,FRB \quad \text{(Rc=1)} \]

<table>
<thead>
<tr>
<th>FRC</th>
<th>FRT</th>
<th>FRB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>63</td>
<td>11</td>
</tr>
<tr>
<td>6</td>
<td>21</td>
<td>26</td>
</tr>
</tbody>
</table>

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.,

\[
\text{ABS}(\text{estimate} - 1/\text{x}) \leq \frac{1}{256}
\]

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>ZX</td>
</tr>
<tr>
<td>(+\frac{1}{2})</td>
<td>(+0)</td>
<td>None</td>
</tr>
<tr>
<td>SNaN</td>
<td>QNaN(^2)</td>
<td>VXSQAN</td>
</tr>
<tr>
<td>QNaN</td>
<td>QNaN</td>
<td>None</td>
</tr>
</tbody>
</table>

\(^1\) No result if FPSCR\(_{VE}\) = 1.
\(^2\) No result if FPSCR\(_{VE}\) = 1.

FPSCR\(_{PFRF}\) is set to the class and sign of the result, except for Invalid Operation Exceptions when FPSCR\(_{VE}\)=1 and Zero Divide Exceptions when FPSCR\(_{ZE}\)=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) FI (undefined)
FX OX UX ZX XX (undefined)
VXSQAN
CR1 (if Rc=1)
Chapter 4. Floating-Point Facility

Floating Reciprocal Square Root Estimate

[Single] A-form

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.

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 the square root of (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 ^3</td>
<td>ZX</td>
</tr>
<tr>
<td>+0</td>
<td>+\infty ^3</td>
<td>ZX</td>
</tr>
<tr>
<td>+\infty</td>
<td>+0</td>
<td>None</td>
</tr>
<tr>
<td>SNaN</td>
<td>QNaN^2</td>
<td>VXSQAN</td>
</tr>
<tr>
<td>QNaN</td>
<td>QNaN</td>
<td>None</td>
</tr>
<tr>
<td>^1 No result if FPSCR_ZE = 1.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>^2 No result if FPSCR_VE = 1.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

FPSCR\_FRP is set to the class and sign of the result, except for Invalid Operation Exceptions when FPSCR\_VE=1 and Zero Divide Exceptions when FPSCR\_ZE=1.

The results of executing this instruction may vary between implementations, and between different executions on the same implementation.

Special Registers Altered:

- FPFR, FR (undefined)
- FX, OX, UX, ZK, XX (undefined)
- VXSQAN, VXSQRT
- CR1 (if Rc=1)

**Note**

See the Notes that appear with *fr*\[s].

---

**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**

[Single] A-form

\[
\text{frsqrte} \quad \text{FRT,FRB} \quad (\text{Rc}=0) \\
\text{frsqrte} \quad \text{FRT,FRB} \quad (\text{Rc}=1) \\
\text{frsqrtes} \quad \text{FRT,FRB} \quad (\text{Rc}=0) \\
\text{frsqrtes} \quad \text{FRT,FRB} \quad (\text{Rc}=1)
\]

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 the square root of (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 ^3</td>
<td>ZX</td>
</tr>
<tr>
<td>+0</td>
<td>+\infty ^3</td>
<td>ZX</td>
</tr>
<tr>
<td>+\infty</td>
<td>+0</td>
<td>None</td>
</tr>
<tr>
<td>SNaN</td>
<td>QNaN^2</td>
<td>VXSQAN</td>
</tr>
<tr>
<td>QNaN</td>
<td>QNaN</td>
<td>None</td>
</tr>
<tr>
<td>^1 No result if FPSCR_ZE = 1.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>^2 No result if FPSCR_VE = 1.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

FPSCR\_FRP is set to the class and sign of the result, except for Invalid Operation Exceptions when FPSCR\_VE=1 and Zero Divide Exceptions when FPSCR\_ZE=1.

The results of executing this instruction may vary between implementations, and between different executions on the same implementation.

**Special Registers Altered:**

- FPFR, FR (undefined)
- FX, OX, UX, ZK, XX (undefined)
- VXSQAN, VXSQRT
- CR1 (if Rc=1)

**Note**

See the Notes that appear with *fr*\[s].
**Floating Test for software Divide X-form**

ftdiv BF,FRA,FRB

<table>
<thead>
<tr>
<th></th>
<th>63</th>
<th>6</th>
<th>9</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>128</th>
</tr>
</thead>
</table>

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.

$fe\_flag$ 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 $fe\_flag$ is set to 0.

$fg\_flag$ 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 $fg\_flag$ is set to 0.

If the implementation guarantees a relative error of fre 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**

ftdiv and 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.

---

**Floating Test for software Square Root X-form**

ftsqrt BF,FRB

<table>
<thead>
<tr>
<th></th>
<th>63</th>
<th>6</th>
<th>9</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>160</th>
</tr>
</thead>
</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 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

---
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 [Single] A-form

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>(Rc=0)</th>
<th>(Rc=1)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fmadd</code></td>
<td>FRT,FRA,FRC,FRB</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>fmadd.</code></td>
<td>FRT,FRA,FRC,FRB</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>fmadds</code></td>
<td>FRT,FRA,FRC,FRB</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>fmadds.</code></td>
<td>FRT,FRA,FRC,FRB</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The operation

\[
\text{FRT} \leftarrow \left[ (\text{FRA}) \times (\text{FRC}) \right] + (\text{FRB})
\]

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 the Floating-Point Rounding Control field RN of the FPSCR and placed into register FRT.

FPSCR\(_{\text{FPRF}}\) is set to the class and sign of the result, except for Invalid Operation Exceptions when FPSCR\(_{\text{VE}}\)=1.

**Special Registers Altered:**

- FPRF FR FI
- FX OX UX XX
- VXSNAN VXISI VXIMZ
- CR1 (if Rc=1)

### Floating Multiply-Subtract [Single] A-form

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>(Rc=0)</th>
<th>(Rc=1)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fmsub</code></td>
<td>FRT,FRA,FRC,FRB</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>fmsub.</code></td>
<td>FRT,FRA,FRC,FRB</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>fmsubs</code></td>
<td>FRT,FRA,FRC,FRB</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>fmsubs.</code></td>
<td>FRT,FRA,FRC,FRB</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The operation

\[
\text{FRT} \leftarrow \left[ (\text{FRA}) \times (\text{FRC}) \right] - (\text{FRB})
\]

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 the Floating-Point Rounding Control field RN of the FPSCR and placed into register FRT.

FPSCR\(_{\text{FPRF}}\) is set to the class and sign of the result, except for Invalid Operation Exceptions when FPSCR\(_{\text{VE}}\)=1.

**Special Registers Altered:**

- FPRF FR FI
- FX OX UX XX
- VXSNAN VXISI VXIMZ
- CR1 (if Rc=1)
Floating Negative Multiply-Add [Single] A-form

\[
\text{fnmadd} \quad \text{FRT,FRA,FRC,FRB} \quad (Rc=0) \\
\text{fnmadd} . \quad \text{FRT,FRA,FRC,FRB} \quad (Rc=1)
\]

\[
\begin{array}{c|cccc|c|c}
0 & 63 & 6 & 11 & 16 & 21 & 26 & 31 \\
\end{array}
\]

\[
\begin{array}{c|cccc|c|c}
63 & 6 & 11 & 16 & 21 & 26 & 31 \\
\end{array}
\]

The operation \( \text{FRT} \leftarrow - (\text{FRA} \times \text{FRC}) + \text{FRB} \) 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 the Floating-Point Rounding Control field RN of the FPSCR, 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.

FPSCR \(_{\text{FPRF}}\) is set to the class and sign of the result, except for Invalid Operation Exceptions when FPSCR \(_{\text{VE}}\) = 1.

Special Registers Altered:
- FPRF, FR, FI
- FX, OX, UX, XX
- VXSNAN, VXISI, VXIMZ
- CR1 (if Rc=1)

Floating Negative Multiply-Subtract [Single] A-form

\[
\text{fnmsub} \quad \text{FRT,FRA,FRC,FRB} \quad (Rc=0) \\
\text{fnmsub} . \quad \text{FRT,FRA,FRC,FRB} \quad (Rc=1)
\]

\[
\begin{array}{c|cccc|c|c}
0 & 63 & 6 & 11 & 16 & 21 & 26 & 31 \\
\end{array}
\]

\[
\begin{array}{c|cccc|c|c}
63 & 6 & 11 & 16 & 21 & 26 & 31 \\
\end{array}
\]

The operation \( \text{FRT} \leftarrow - (\text{FRA} \times \text{FRC}) - \text{FRB} \) 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 the Floating-Point Rounding Control field RN of the FPSCR, 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.

FPSCR \(_{\text{FPRF}}\) is set to the class and sign of the result, except for Invalid Operation Exceptions when FPSCR \(_{\text{VE}}\) = 1.

Special Registers Altered:
- FPRF, FR, FI
- FX, OX, UX, XX
- VXSNAN, VXISI, VXIMZ
- CR1 (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**

```
frsp: FRT,FRB (Rc=0)
frsp. FRT,FRB (Rc=1)
```

The floating-point operand in register FRB is rounded to single-precision, using the rounding mode specified by RN, and placed into register FRT.

The rounding is described fully in Section A.1, “Floating-Point Round to Single-Precision Model” on page 775.

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
- XX
- VXSNAN
- CR1 (if 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**

```
fcxid: FRT,FRB (Rc=0)
fctid. FRT,FRB (Rc=1)
```

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 specified by RN.

If the rounded value is greater than 263-1, then the result is 0x7FFF_FFFF_FFFF_FFFF and VXCVI is set to 1.

Otherwise, if the rounded value is less than -263, 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 779.

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 Signed Doubleword format X-form

\[
fctidz \ FRT,FRB \quad (Rc=0)
\]
\[
fctidz. \ FRT,FRB \quad (Rc=1)
\]

Let \( src \) be the double-precision floating-point value in \( FRB \).

If \( src \) is a NaN, then the result is \( 0x8000_0000_0000_0000 \), \( VXCVI \) 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 779.

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:
\[
\begin{array}{llllllll}
FPRF & (undefined) & FR & FI \\
FX & XX & VXSNAN & VXCVI \\
CR1 & (if Rc=1)
\end{array}
\]

Floating Convert with round
Double-Precision To Unsigned Doubleword format X-form

\[
fctidu \ FRT,FRB \quad (Rc=0)
\]
\[
fctidu. \ FRT,FRB \quad (Rc=1)
\]

Let \( src \) be the double-precision floating-point value in \( FRB \).

If \( src \) is a NaN, then the result is \( 0x0000_0000_0000_0000 \), \( VXCVI \) 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 779.

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:
\[
\begin{array}{llllllll}
FPRF & (undefined) & FR & FI \\
FX & XX & VXSNAN & VXCVI \\
CR1 & (if Rc=1)
\end{array}
\]
Floating Convert with truncate
Double-Precision To Unsigned Doubleword format X-form

fctiduz FRT,FRB \((Rc=0)\)
fctiduz FRT,FRB \((Rc=1)\)

Let \(src\) be the double-precision floating-point value in \(FRB\).

If \(src\) is a NaN, then the result is \(0\times0000_0000_0000_0000\), \(VXCVI\) 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 \(2^{64}-1\), 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 779.

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:

\[
\begin{align*}
FPRF & \text{ (undefined)} \quad FR \quad FI \\
FX & \text{ (if } Rc = 1 \text{)} \quad VXSNAN \quad VXCVI \\
CR1 & \text{ (if } Rc = 1 \text{)}
\end{align*}
\]

Floating Convert with round
Double-Precision To Signed Word format X-form

fctiw FRT,FRB \((Rc=0)\)
fctiw FRT,FRB \((Rc=1)\)

Let \(src\) be the double-precision floating-point value in \(FRB\).

If \(src\) is a NaN, then the result is \(0x8000_0000\), \(VXCVI\) 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 \(2^{31}-1\), then the result is \(0x7FFF_FFFF\), and \(VXCVI\) is set to 1.

Otherwise, if the rounded value is less than \(-2^{31}\), then 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 \(XX\) 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 779.

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:

\[
\begin{align*}
FPRF & \text{ (undefined)} \quad FR \quad FI \\
FX & \text{ (if } Rc = 1 \text{)} \quad VXSNAN \quad VXCVI \\
CR1 & \text{ (if } Rc = 1 \text{)}
\end{align*}
\]
Floating Convert with truncate
Double-Precision To Signed Word format X-form

fctiwz  FRT,FRB  
(Rc = 0)
fctiwz. FRT,FRB  
(Rc = 1)

Let src be the double-precision floating-point value in FRB.

If src is a NaN, then the result is 0x8000_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^{31}, then the result is 0x7FFF_FFFF, and VXCVI is set to 1.
Otherwise, if the rounded value is less than 2^{31}, then 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 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 FRT32:63 and FRT0:31 is undefined.

The conversion is described fully in Section A.2, "Floating-Point Convert to Integer Model" on page 779.

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:

<table>
<thead>
<tr>
<th>FPRF (undefined)</th>
<th>FR</th>
<th>FI</th>
</tr>
</thead>
<tbody>
<tr>
<td>FX, XX</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VXSNAN, VXCVI</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CR1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Floating Convert with round
Double-Precision To Unsigned Word format X-form

fctiwu  FRT,FRB  
(Rc = 0)
fctiwu. FRT,FRB  
(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 specified by RN.

If the rounded value is greater than 2^{32}, 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 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 FRT32:63 and FRT0:31 is undefined.

The conversion is described fully in Section A.2, "Floating-Point Convert to Integer Model" on page 779.

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:

<table>
<thead>
<tr>
<th>FPRF (undefined)</th>
<th>FR</th>
<th>FI</th>
</tr>
</thead>
<tbody>
<tr>
<td>FX, XX</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VXSNAN, VXCVI</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CR1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(Rc = 1)
### Floating Convert with truncate

**Double-Precision To Unsigned Word format X-form**

`fctiwuz FRT,FRB`  \((Rc=0)\)

`fctiwuz FRT,FRB`  \((Rc=1)\)

<table>
<thead>
<tr>
<th></th>
<th>63</th>
<th>FRT</th>
<th></th>
<th>FRB</th>
<th></th>
<th>143</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>8</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td></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`, `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 `FRT32:63` and `FRT0:31` is undefined.

The conversion is described fully in Section A.2, “Floating-Point Convert to Integer Model” on page 779.

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**

`fcfid FRT,FRB`  \((Rc=0)\)

`fcfid FRT,FRB`  \((Rc=1)\)

<table>
<thead>
<tr>
<th></th>
<th>63</th>
<th>FRT</th>
<th></th>
<th>FRB</th>
<th></th>
<th>846</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>8</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td></td>
<td>31</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 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

\[ \text{fcfidu} \quad \text{FRT,FRB} \quad (Rc=0) \]
\[ \text{fcfidu.} \quad \text{FRT,FRB} \quad (Rc=1) \]

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 double-precision, using the rounding mode specified by FPSCR_{RN}, and placed into register FRT.

The conversion is described fully in Section A.3, "Floating-Point Convert from Integer Model".

FPSCR_{FPRF} is set to the class and sign of the result. FR is set if the result is incremented when rounded. FPSCR_{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 double-precision floating-point can be accomplished by loading the word from storage using \text{Load Float Word and Zero Indexed} and then using \text{fcfidu}.

Floating Convert with round Signed Doubleword to Single-Precision format X-form

\[ \text{fcfids} \quad \text{FRT,FRB} \quad (Rc=0) \]
\[ \text{fcfids.} \quad \text{FRT,FRB} \quad (Rc=1) \]

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 single-precision, using the rounding mode specified by FPSCR_{RN}, and placed into register FRT.

The conversion is described fully in Section A.3, "Floating-Point Convert from Integer Model".

FPSCR_{FPRF} is set to the class and sign of the result. FR is set if the result is incremented when rounded. FPSCR_{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 single-precision floating-point can be accomplished by loading the word from storage using \text{Load Float Word Algebraic Indexed} and then using \text{fcfids}.
Floating Convert with round Unsigned Doubleword to Single-Precision format X-form

<table>
<thead>
<tr>
<th>fcfidus</th>
<th>FRT, FRB</th>
<th>(Rc=0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>fcfidus</td>
<td>FRT, FRB</td>
<td>(Rc=1)</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 single-precision, using the rounding mode specified by FPSCR_{RN}, and placed into register FRT.

The conversion is described fully in Section A.3, “Floating-Point Convert from Integer Model”.

FPSCR_{FPRF} is set to the class and sign of the result. FR is set if the result is incremented when rounded. FPSCR_{FI} is set if the result is inexact.

Special Registers Altered:
- FPRF, FR, FI
- FX, XX
- CR1

Programming Note

Converting a 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 784.

Programming Note

These instructions set FPSCR_{FR, FI} to 0b00 regardless of whether the result is inexact or rounded because there is a desire to preserve the value of FPSCR_{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

frin. FRT,FRB (Rc=0)
frin. FRT,FRB (Rc=1)

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.

FPSCR_{FPRF} is set to the class and sign of the result, except for Invalid Operation Exceptions when FPSCR_{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

frip. FRT,FRB (Rc=0)
frip. FRT,FRB (Rc=1)

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.

FPSCR_{FPRF} is set to the class and sign of the result, except for Invalid Operation Exceptions when FPSCR_{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

friz. FRT,FRB (Rc=0)
friz. FRT,FRB (Rc=1)

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.

FPSCR_{FPRF} is set to the class and sign of the result, except for Invalid Operation Exceptions when FPSCR_{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

frim FRT,FRB (Rc=0)
frim FRT,FRB (Rc=1)

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.

FPSCR_{FPRF} is set to the class and sign of the result, except for Invalid Operation Exceptions when FPSCR_{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.

The CR field and the FPCC are set as follows:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>FL</td>
<td>(FRA) &lt; (FRB)</td>
</tr>
<tr>
<td>1</td>
<td>FG</td>
<td>(FRA) &gt; (FRB)</td>
</tr>
<tr>
<td>2</td>
<td>FE</td>
<td>(FRA) = (FRB)</td>
</tr>
<tr>
<td>3</td>
<td>FU</td>
<td>(FRA) ? (FRB) (unordered)</td>
</tr>
</tbody>
</table>

Floating Compare Unordered X-form

\[ \text{fcmpu BF,FRA,FRB} \]

\[
\begin{array}{cccccc}
0 & 63 & 6 & 5 & 4 & 3 \\
& BF & // & FRA & FRB & 0 \\
\end{array}
\]

If (FRA) is a NaN or
(FRB) is a NaN then c ← 0b0001
else if (FRA) < (FRB) then c ← 0b1000
else if (FRA) > (FRB) then c ← 0b0100
else c ← 0b0010

FPCC ← c

CR4×BF:4×BF+3 ← c

if (FRA) is an SNaN or
(FRB) is an SNaN then
VXSNAN ← 1

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

\[ \text{fcmpo BF,FRA,FRB} \]

\[
\begin{array}{cccccc}
0 & 63 & 6 & 5 & 4 & 3 & 32 \\
& BF & // & FRA & FRB & 0 & 31 \\
\end{array}
\]

If (FRA) is a NaN or
(FRB) is a NaN then c ← 0b0001
else if (FRA) < (FRB) then c ← 0b1000
else if (FRA) > (FRB) then c ← 0b0100
else c ← 0b0010

FPCC ← c

CR4×BF:4×BF+3 ← c

if (FRA) is an SNaN or
(FRB) is an SNaN then
VXSNAN ← 1

if VE = 0 then VXVC ← 1
else if (FRA) is a QNaN or
(FRB) is a QNaN then VXVC ← 1

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 and, 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

\[ \text{fsel} \quad FRT,FRA,FRC,FRB \quad (Rc=0) \]
\[ \text{fsel} \quad FRT,FRA,FRC,FRB \quad (Rc=1) \]

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

Examples of uses of this instruction can be found in Sections E.2, “Floating-Point Conversions” on page 642 and E.3, “Floating-Point Selection” on page 646.

Warning: Care must be taken in using \texttt{fsel} if IEEE compatibility is required, or if the values being tested can be NaNs or infinities; see Section E.3.4, “Notes” on page 646.

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 \texttt{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.

Warnings: Care must be taken in using \texttt{fsel} if IEEE compatibility is required, or if the values being tested can be NaNs or infinities; see Section E.3.4, “Notes” on page 646.

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>\text{fsel} 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>\text{fneg} fs,fa \text{fsel} fx,fs,fy,fz</td>
<td>(1,2)</td>
</tr>
<tr>
<td>if ( a = 0.0 ) then ( x \leftarrow y ) else ( x \leftarrow z )</td>
<td>\text{fsel} fx,fa,fy,fz \text{fneg} fs,fa \text{fsel} fx,fs,fy,fz</td>
<td>(1)</td>
</tr>
</tbody>
</table>

<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>\text{fsub} fs,fa,fb \text{fsel} 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>\text{fsub} fs,fb,fa \text{fsel} fx,fs,fy,fz</td>
<td>(3,4,5)</td>
</tr>
<tr>
<td>if ( a = b ) then ( x \leftarrow y ) else ( x \leftarrow z )</td>
<td>\text{fsub} fs,fa,fb \text{fsub} fx,fs,fy,fz \text{fneg} fs,fs \text{fsel} fx,fs,fy,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 \texttt{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 \texttt{fcmpu} and \texttt{Branch Conditional} instructions instead of \texttt{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 \(VXI S\) 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 for **mffsce**, **mffscdrn[i]**, **mffscrn[i]**, and **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.

(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 $4^*k: 4^*k+3$.

#### Move From FPSCR [& Clear Enables | Lightweight | Control [ & Set (DRN|RN) [Immediate]]] X-form

<table>
<thead>
<tr>
<th>Instruction</th>
<th>FPSCR</th>
<th>CR Field (Rc=0/1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>mffs FRT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mffs. FRT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mffsce FRT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mffscdrn FRT,FRB</td>
<td>63</td>
<td>0 0 0 16 21 583</td>
</tr>
<tr>
<td>mffscrn FRT,FRB</td>
<td>63</td>
<td>0 2 6 16 21 583</td>
</tr>
<tr>
<td>mffscrn FRT,RM</td>
<td>63</td>
<td>0 2 7 16 21 583</td>
</tr>
<tr>
<td>mffsl FRT</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For **Move From FPSCR (mffs[i])**, do the following.

The contents of the FPSCR are placed into register FRT.

If Rc=1, CR field 1 is set to the value $FX||EX||UX||OX$.

For **Move From FPSCR & Clear Enables (mffsce)**, do the following.

The contents of the FPSCR are placed into register FRT.

The contents of bits 56:60 (VE, OE, UE, ZE, XE) of the FPSCR are set to 0.

For **Move From FPSCR Control & set DRN (mffscdrn)**, do the following.

Let new_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 (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.

\[ \text{new}_R N \] is placed into bits 29:31 of the FPSCR (DRN).

For \textit{Move From FPSCR Control \& set DRN Immediate (mffscdrni)}, do the following.

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.

For \textit{Move From FPSCR Control \& set RN (mffscrn)}, do the following.

Let \[ \text{new}_R N \] 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.

\[ \text{new}_R N \] is placed into bits 62:63 of the FPSCR (RN).

For \textit{Move From FPSCR Control \& set RN Immediate (mffscrni)}, do the following.

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 62:63 of the FPSCR (RN) are set to the value of RM.

For \textit{Move From FPSCR Lightweight (mffsl)}, do the following.

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.

\textbf{Special Registers Altered:}

\[ \text{CR1} \quad \text{(if } R_c = 1 \text{)} \]

\textbf{Programming Note}

\textit{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.

\textit{mffscdrni} and \textit{mffscrn} permit software to simultaneously read control bits in the FPSCR and set either the DRN or RN fields without the higher latency typically associated with accessing the status bits.

\textbf{Move to Condition Register from FPSCR X-form}

\begin{tabular}{|c|c|c|c|c|c|c|c|}
\hline
0 & 63 & BF & // & BFA & // & 64 & //
\hline
6 & 9 & 11 & 14 & 16 & 21 & 31 &
\hline
\end{tabular}

The contents of FPSCR\textsubscript{32:63} 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.

\textbf{Special Registers Altered:}

\begin{itemize}
  \item CR field BF
  \item FX OX \quad \text{(if } BFA = 0 \text{)}
  \item UX ZX XX VXSNAN \quad \text{(if } BFA = 1 \text{)}
  \item VXISI VXIDI VXZDZ VXIMZ \quad \text{(if } BFA = 2 \text{)}
  \item VXVC \quad \text{(if } BFA = 3 \text{)}
  \item VXSOFT VXSQRT VXCVI \quad \text{(if } BFA = 5 \text{)}
\end{itemize}
**Move To FPSCR Field Immediate X-form**

mtfsfi BF,U,W (Rc=0)
mtfsfi. BF,U,W (Rc=1)

The value of the U field is placed into FPSCR field \(BF+8*(1-W)\).

FPSCR\(_{FX}\) is altered only if BF = 0 and W = 0.

**Special Registers Altered:**
- FPSCR field BF + 8\(^*(1-W)\)
- CR1 (if Rc=1)

---

**Programming Note**

`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.

---

**Move To FPSCR Fields XFL-form**

mtfsf FLM,FRB,L,W (Rc=0)
mtfsf. FLM,FRB,L,W (Rc=1)

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.

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 FPSCR\(_{32:35}\) 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 125, and not from U\(_{1:2}\).
### Move To FPSCR Bit 0 X-form

```
<table>
<thead>
<tr>
<th>mtsb0</th>
<th>BT</th>
<th></th>
<th></th>
<th>63</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>30</th>
</tr>
</thead>
</table>
```

Bit BT+32 of the FPSCR is set to 0.

**Special Registers Altered:**
- FPSCR Bit BT+32
- CR1 (if Rc=1)

**Programming Note**
Bits 33 and 34 (FEX and VX) cannot be explicitly reset.

---

### Move To FPSCR Bit 1 X-form

```
<table>
<thead>
<tr>
<th>mtsb1</th>
<th>BT</th>
<th></th>
<th></th>
<th>63</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>30</th>
</tr>
</thead>
</table>
```

Bit BT+32 of the FPSCR is set to 1.

**Special Registers Altered:**
- FPSCR bits BT+32 and FX
- CR1 (if Rc=1)

**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 unsinged 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 193.

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)
  - \( \infty \times \infty \) (VXISI)
  - \( \infty \div \infty \) (VXIDI)
  - 0 \( \div \) 0 (VXZDZ)
  - \( \infty \%\) 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 185 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.

5.3 Floating-Point Exception Summary

- Floating-Point Move instructions can be used to move operands between FPRs.

The bit definitions for the FPSCR are as follows.

<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></td>
<td>See Section 5.5.2, “Rounding Mode Specification” on page 183.</td>
</tr>
<tr>
<td></td>
<td>000 Round to Nearest, Ties to Even</td>
</tr>
<tr>
<td></td>
<td>001 Round toward Zero</td>
</tr>
<tr>
<td></td>
<td>010 Round toward +Infinity</td>
</tr>
<tr>
<td></td>
<td>011 Round toward -Infinity</td>
</tr>
<tr>
<td></td>
<td>100 Round to Nearest, Ties away from 0</td>
</tr>
<tr>
<td></td>
<td>101 Round to Nearest, Ties toward 0</td>
</tr>
<tr>
<td></td>
<td>110 Round to away from Zero</td>
</tr>
<tr>
<td></td>
<td>111 Round to Prepare for Shorter Precision</td>
</tr>
</tbody>
</table>

Floating-Point Exception Summary (FX)

- Every floating-point instruction, except mtfsli and mttsf, implicitly sets FPSCRFX to 1 if that instruction causes any of the floating-point exception bits in the FPSCR to change from 0 to 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, mtfsli, mttsf, mtfsb0, and mtfsb1 cannot alter FPSCRFX explicitly.

Floating-Point Invalid Operation Exception Summary (VX)

- 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 \texttt{FPSCRXX} explicitly.

\textbf{Floating-Point Overflow Exception} (OX) 
See Section 5.5.10.3, “Overflow Exception” on page 189.

\textbf{Floating-Point Underflow Exception} (UX) 
See Section 5.5.10.4, “Underflow Exception” on page 189.

\textbf{Floating-Point Zero Divide Exception} (ZX) 
See Section 5.5.10.2, “Zero Divide Exception” on page 188.

\textbf{Floating-Point Inexact Exception} (XX) 
See Section 5.5.10.5, “Inexact Exception” on page 190.

\textbf{Floating-Point Invalid Operation Exception} (VXSNAN) 
See Section 5.5.10.1, “Invalid Operation Exception” on page 187.

\textbf{Floating-Point Invalid Operation Exception} ({\(\frac{1}{2} - \frac{1}{2}\)}) (VXISI) 
See Section 5.5.10.1.

\textbf{Floating-Point Invalid Operation Exception} ({\(\frac{1}{2} + \frac{1}{2}\)}) (VXIDI) 
See Section 5.5.10.1.

\textbf{Floating-Point Invalid Operation Exception} (0 + 0) (VXZDZ) 
See Section 5.5.10.1.

\textbf{Floating-Point Invalid Operation Exception} (Invalid Compare) (VXVC) 
See Section 5.5.10.1.

\textbf{Floating-Point Fraction Rounded} (FR) 
The last Arithmetic or Rounding and Conversion instruction incremented the fraction during rounding. See Section 5.5.1, “Rounding” on page 182. This bit is not sticky.

\textbf{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 5.5.1. This bit is not sticky.

\textbf{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.

\textbf{Floating-Point Result Class Descriptor} (C) 
Arithmetic, rounding, and conversion instructions may set this bit with the FPCC bits, to indicate the class of the result as shown in Figure 58 on page 178.

\textbf{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 178. 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.

\textbf{Floating-Point Less Than or Negative} (FL or <) 

\textbf{Floating-Point Greater Than or Positive} (FG or >) 

\textbf{Floating-Point Equal or Zero} (FE or =) 

\textbf{Floating-Point Unordered or NaN} (FU or ?) 

Reserved

\textbf{Floating-Point Invalid Operation Exception} (Software Request) (VXSOFT) 
This bit can be altered only by \texttt{mcrfs}, \texttt{mtfsfi}, \texttt{mtfsf}, \texttt{mtfsb0}, or \texttt{mtfsb1}. See Section 5.5.10.1, “Invalid Operation Exception” on page 187. Neither used nor changed by DFP.

\textbf{Programming Note} 
Although the architecture does not provide a DFP square root instruction, if software simulates such an instruction, it should set bit 54 whenever the source operand of the square root function is invalid.

\textbf{Floating-Point Invalid Operation Exception} (Invalid Conversion) (VXCVI) 
See Section 5.5.10.1.
56     Floating-Point Invalid Operation Exception Enable (VE)
See Section 5.5.10.1.
57     Floating-Point Overflow Exception Enable (OE)
See Section 5.5.10.3, “Overflow Exception” on page 189.
58     Floating-Point Underflow Exception Enable (UE)
See Section 5.5.10.4, “Underflow Exception” on page 189.
59     Floating-Point Zero Divide Exception Enable (ZE)
See Section 5.5.10.2, “Zero Divide Exception” on page 188.
60     Floating-Point Inexact Exception Enable (XE)
See Section 5.5.10.5, “Inexact Exception” on page 190.
61 Reserved (not used by DFP)
62:63  Binary Floating-Point Rounding Control (RN)
See Section 5.5.1, “Rounding” on page 182.
  00 Round to Nearest
  01 Round toward Zero
  10 Round toward +Infinity
  11 Round toward -Infinity

<table>
<thead>
<tr>
<th>Result Flags</th>
<th>Result Value Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>C  &lt;  &gt;  =  ?</td>
<td></td>
</tr>
<tr>
<td>0 0 0 0 1</td>
<td>Signaling NaN (DFP only)</td>
</tr>
<tr>
<td>1 0 0 0 1</td>
<td>Quiet NaN</td>
</tr>
<tr>
<td>0 1 0 0 1</td>
<td>- Infinity</td>
</tr>
<tr>
<td>0 1 0 0 0</td>
<td>- Normal Number</td>
</tr>
<tr>
<td>1 1 0 0 0</td>
<td>- Subnormal 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>+ Subnormal Number</td>
</tr>
<tr>
<td>0 0 1 0 0</td>
<td>+ Normal Number</td>
</tr>
<tr>
<td>0 0 1 0 1</td>
<td>+ Infinity</td>
</tr>
</tbody>
</table>

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 (D) as shown in Figure 59.

Figure 58. Floating-Point Result Flags

Figure 59. Format for Unsigned Decimal Data

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

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 1110 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-opera-
tion exception occurs. A summary of digit and sign codes are provided in Figure 61.

<table>
<thead>
<tr>
<th>Binary Code</th>
<th>Recognized As</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
</tr>
<tr>
<td>0010</td>
<td>2</td>
</tr>
<tr>
<td>0011</td>
<td>3</td>
</tr>
<tr>
<td>0100</td>
<td>4</td>
</tr>
<tr>
<td>0101</td>
<td>5</td>
</tr>
<tr>
<td>0110</td>
<td>6</td>
</tr>
<tr>
<td>0111</td>
<td>7</td>
</tr>
<tr>
<td>1000</td>
<td>8</td>
</tr>
<tr>
<td>1001</td>
<td>9</td>
</tr>
<tr>
<td>1010</td>
<td>Invalid</td>
</tr>
<tr>
<td>1011</td>
<td>Invalid</td>
</tr>
<tr>
<td>1100</td>
<td>Invalid</td>
</tr>
<tr>
<td>1101</td>
<td>Invalid</td>
</tr>
<tr>
<td>1110</td>
<td>Invalid</td>
</tr>
<tr>
<td>1111</td>
<td>Invalid</td>
</tr>
</tbody>
</table>

Figure 61. Summary of BCD Digit and Sign Codes

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 \(\%\) \(10^5\) and 10 \(\%\) \(10^{10}\) 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:

![Figure 62. DFP Short format](image)

![Figure 63. DFP Long format](image)

![Figure 64. DFP Extended format](image)

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>$G_{0:4}$</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 significand 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>Format</th>
<th>DFP Short</th>
<th>DFP Long</th>
<th>DFP Extended</th>
</tr>
</thead>
<tbody>
<tr>
<td>Widths (bits):</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Format</td>
<td>32</td>
<td>64</td>
<td>128</td>
</tr>
<tr>
<td>Sign ($S$)</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Combination ($G$)</td>
<td>11</td>
<td>13</td>
<td>17</td>
</tr>
<tr>
<td>Trailing Significand ($T$)</td>
<td>20</td>
<td>50</td>
<td>110</td>
</tr>
<tr>
<td>Exponent:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum biased</td>
<td>191</td>
<td>767</td>
<td>12,287</td>
</tr>
<tr>
<td>Maximum ($X_{max}$)</td>
<td>90</td>
<td>369</td>
<td>6111</td>
</tr>
<tr>
<td>Minimum ($X_{min}$)</td>
<td>-101</td>
<td>-398</td>
<td>-6176</td>
</tr>
<tr>
<td>Bias</td>
<td>101</td>
<td>398</td>
<td>6176</td>
</tr>
<tr>
<td>Precision ($p$) (digits)</td>
<td>7</td>
<td>16</td>
<td>34</td>
</tr>
<tr>
<td>Magnitude:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum normal number ($N_{max}$)</td>
<td>$(10^7 - 1) \times 10^{90}$</td>
<td>$(10^{16} - 1) \times 10^{369}$</td>
<td>$(10^{34} - 1) \times 10^{6111}$</td>
</tr>
<tr>
<td>Minimum normal number ($N_{min}$)</td>
<td>$1 \times 10^{-95}$</td>
<td>$1 \times 10^{-383}$</td>
<td>$1 \times 10^{-6143}$</td>
</tr>
<tr>
<td>Minimum subnormal number ($D_{min}$)</td>
<td>$1 \times 10^{-101}$</td>
<td>$1 \times 10^{-398}$</td>
<td>$1 \times 10^{-6176}$</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 FPSCR_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

The following figure shows the encoding of NaN and infinity data classes.

<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>111111xx . . . 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

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 FRA and an SNaN in FRB, the SNaN in FRB is propagated.
- Otherwise, propagate the NaN in FRA.

### 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 183.)

### 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 FR and FI. When an inexact exception occurs, FI is set to one; otherwise, FI is set to zero. When an inexact exception occurs and if the rounded result is greater in magnitude than the intermediate result, then FR is set to one; otherwise, FR is set to zero. The exception is the *Round to FP Integer Without Inexact* instruction, which always sets FR and FI 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 Z1 or Z2 is chosen to approximate the result, where Z1 and Z2 are the next larger and smaller permissible resultant values, respectively.

#### Figure 70. Rounding

- **Round to Nearest, Ties to Even**
  Choose the value that is closer to Z (Z1 or Z2). 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 \(N_{\text{max}} + 0.5Q(N_{\text{max}})\) is rounded to infinity with no change in sign; where \(Q(N_{\text{max}})\) is the quantum of \(N_{\text{max}}\).

- **Round toward 0**
  Choose the smaller in magnitude (Z1 or Z2).

- **Round toward +∞**
  Choose Z1.

- **Round toward -∞**
  Choose Z2.

- **Round to Nearest, Ties away from 0**
  Choose the value that is closer to Z (Z1 or Z2). In case of a tie, choose the larger in magnitude (Z1 or Z2). However, an infinitely precise result with magnitude at least \(N_{\text{max}} + 0.5Q(N_{\text{max}})\) is rounded to infinity with no change in sign; where \(Q(N_{\text{max}})\) is the quantum of \(N_{\text{max}}\).

- **Round to Nearest, Ties toward 0**
  Choose the value that is closer to Z (Z1 or Z2). In case of a tie, choose the smaller in magnitude (Z1 or Z2). However, an infinitely precise result with magnitude
greater than \((N_{\text{max}} + 0.5Q(N_{\text{max}}))\) is rounded to infinity with no change in sign; where \(Q(N_{\text{max}})\) is the quantum of \(N_{\text{max}}\).

**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 FPSCR_{DRN}</td>
</tr>
</tbody>
</table>

**Figure 72. Primary 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 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 73. Secondary Encoding of Rounding-Mode Control**

**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 FPSCR\_{\text{VX-SNAN}} flag, but Compare Ordered recognizes the exception and sets both the FPSCR\_{\text{VXSNAN}} and FPSCR\_{\text{VXVC}} flags. In case 2, Compare Unordered does not recognize an exception, but Compare Ordered recognizes an invalid-operation exception and sets the FPSCR\_{\text{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\_{\text{FPCC}} 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 FPSCR\_{\text{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 FPSCR\_{\text{FPCC}} field and CR field BF.

The Test Significance instruction compares the number of significant digits of one source operand with the referenced number of significant digits in another source operand. The test result is indicated in the FPSCR\_{\text{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 an 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**
  
  \[
  \begin{align*}
  \text{SNaN} & \quad \infty \times \infty \\
  \text{SNaN} & \quad \infty \div \infty \\
  0 & \div 0 \\
  \text{SNaN} & \quad \% \infty \\
  \text{Invalid Compare} & \\
  \text{Invalid Conversion} & \\
  \end{align*}
  \]

- **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 \texttt{mtfsi} or \texttt{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) may be set with Invalid Operation Exception (Invalid Compare) for \texttt{Compare Ordered} instructions.
- Invalid Operation Exception (SNaN) may be set with Invalid Operation Exception (Invalid Conversion) for \texttt{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 \( R_c = 1 \)) 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

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 \texttt{Move To FPSCR} instruction causes an exception bit and the corresponding enable bit both to be 1; the \texttt{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, \textit{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><strong>Ignore Exceptions Mode</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>DFP 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><strong>Imprecise Nonrecoverable Mode</strong></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. 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><strong>Imprecise Recoverable Mode</strong></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><strong>Precise Mode</strong></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 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 completed, 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 185 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 Significant Left Immediate, and Shift Significant Right Immediate
For add or subtract operations, magnitude subtraction of infinities ($\pm \infty$) + ($\pm \infty$)

Division of infinity by infinity ($\infty \div \infty$)

Division of zero by zero ($0 \div 0$)

Multiplication of infinity by zero ($\infty \times 0$)

The Quantize operation detects that the significand associated with the specified target exponent would have more significant digits than the target-format precision

For the Quantize operation, when one source operand specifies an infinity and the other specifies a finite number

The Reround operation detects that the target exponent associated with the specified target significand would be greater than $X_{\text{max}}$

The Encode BCD To DPD operation detects an invalid BCD digit or sign code

The Convert To Fixed operation involving a number too large in magnitude to be represented in the target format, or involving a NaN.

Programming Note

In addition, an Invalid Operation Exception occurs if software explicitly requests this by executing an `mutsli`, `mutsf`, or `mtfsb1` instruction that sets FPSCR_VXSOFT to 1 (Software Request). The purpose of FPSCR_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.

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 (FPSCR_VE=1) and Invalid Operation occurs, the following actions are taken:

1. One or two Invalid Operation Exceptions are set:
   - FPSCR_VXSNAN (if SNaN)
   - FPSCR_VXISI (if $\infty \div \infty$)
   - FPSCR_VXIDI (if $\infty \times 0$)
   - FPSCR_VXZDZ (if 0 $\div \infty$)
   - FPSCR_VXIMZ (if $\infty \times 0$)
   - FPSCR_VXVC (if invalid comp)
   - FPSCR_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
   - FPSCR_FRFI are set to zero
   - FPSCR_FPRF 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.
     - FPSCR_FRFI are set to zero
     - FPSCR_FPRF is unchanged

4. If the operation is a compare
   - FPSCR_FRFI are unchanged
   - FPSCR_FPCCC 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 (FPSCR_ZE=1) and Zero Divide occurs, the following actions are taken:

1. Zero Divide Exception is set
   - FPSCR_ZX $\leftarrow 1$

2. The target FPR is unchanged
3. FPSCR_FRFI are set to zero
4. FPSCR_FPRF is unchanged

When Zero Divide Exception is disabled (FPSCR_ZE=0) and Zero Divide occurs, the following actions are taken:

1. Zero Divide Exception is set
   - FPSCR_ZX $\leftarrow 1$

2. The target FPR is set to $\pm \infty$, where the sign is determined by the XOR of the signs of the operands
3. \( \text{FPSCR}_{\text{FR, FI}} \) are set to zero
4. \( \text{FPSCR}_{\text{FPRF}} \) is set to indicate the class and sign of the result (\( \pm \infty \))

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 \textit{Reround}, the following describes the handling of the IEEE overflow exception condition. The \textit{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 (\( \text{FPSCR}_{\text{OE}}=1 \)) and overflow occurs, the following actions are taken:

1. Overflow Exception is set
   \( \text{FPSCR}_{\text{OX}} \leftarrow 1 \)
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 \textit{wrapped result}.
   The exponent adjustment for all operations, except for \textit{Round To DFP Short} and \textit{Round To DFP Long}, is 576 for DFP Long and 9216 for DFP Extended. For \textit{Round To DFP Short} and \textit{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 \textit{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. \( \text{FPSCR}_{\text{FPRF}} \) is set to indicate the class and sign of the result (\( \pm \infty \) or \( \pm \text{Normal number} \))

When Overflow Exception is disabled (\( \text{FPSCR}_{\text{OE}}=0 \)) and overflow occurs, the following actions are taken:

1. Overflow Exception is set
   \( \text{FPSCR}_{\text{OX}} \leftarrow 1 \)
2. Inexact Exception is set
   \( \text{FPSCR}_{\text{XX}} \leftarrow 1 \)
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>Plus</th>
<th>Minus</th>
</tr>
</thead>
<tbody>
<tr>
<td>Round to Nearest, Ties to Even</td>
<td>+\infty</td>
<td>-\infty</td>
</tr>
<tr>
<td>Round toward 0</td>
<td>+N_{\text{max}}</td>
<td>-N_{\text{max}}</td>
</tr>
<tr>
<td>Round toward +\infty</td>
<td>+\infty</td>
<td>-N_{\text{max}}</td>
</tr>
<tr>
<td>Round toward -\infty</td>
<td>+N_{\text{max}}</td>
<td>-\infty</td>
</tr>
<tr>
<td>Round to Nearest, Ties away from 0</td>
<td>+\infty</td>
<td>-\infty</td>
</tr>
<tr>
<td>Round to Nearest, Ties toward 0</td>
<td>+\infty</td>
<td>-\infty</td>
</tr>
<tr>
<td>Round away from 0</td>
<td>+\infty</td>
<td>-\infty</td>
</tr>
<tr>
<td>Round to prepare for shorter precision</td>
<td>+N_{\text{max}}</td>
<td>-N_{\text{max}}</td>
</tr>
</tbody>
</table>

5.5.10.4 Underflow Exception

Definition

Except for \textit{Reround}, the following describes the handling of the IEEE underflow exception condition. The \textit{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 (FPSCR[UE]=1) and underflow occurs, the following actions are taken:

1. Underflow Exception is set
   FPSCRUX ← 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. FPSCR[FPRF] is set to indicate the class and sign of the result (± Normal number)

When Underflow Exception is disabled (FPSCR[UE]=0) and underflow occurs, the following actions are taken:

1. Underflow Exception is set
   FPSCRUX ← 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. FPSCR[FPRF] is set to indicate the class and sign of the result (± Normal number, ± Subnormal Number, or ± 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
   FPSCR[XX] ← 1
2. The rounded or overflowed result is placed into the target FPR
3. FPSCR[FPRF] is set to indicate the class and sign of the result

Program 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.

![Table](image)

**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 away from 0.
- Round to Nearest, Ties to even.
- Round to Nearest, Ties toward 0.
- Round toward +∞.
- Round toward -∞.
- Round toward 0.

**Figure 76. Rounding and Range Actions (Part 1)**
| Case          | Is \( r \) inexact (\( r \neq v \)) | OE=1 | UE=1 | XE=1 | Is \( r \) incremented (\( |r|>|v| \)) | Is \( q \) inexact (\( q \neq v \)) | Is \( q \) incremented (\( |q|>|v| \)) | Returned Results and Status Setting* |
|--------------|------------------------------------|------|------|------|--------------------------------------|--------------------------------------|--------------------------------------|--------------------------------------|
| Overflow     | Yes \(^1\)                          | No   | —    | No   | No                                   | —                                    | —                                    | \( T(r), OX \leftarrow 1, FI \leftarrow 1, FR \leftarrow 0, XX \leftarrow 1 \) |
| Overflow     | Yes \(^1\)                          | No   | —    | Yes  | —                                    | —                                    | —                                    | \( T(r), OX \leftarrow 1, FI \leftarrow 1, FR \leftarrow 1, XX \leftarrow 1 \) |
| Overflow     | Yes \(^1\)                          | No   | —    | Yes  | —                                    | —                                    | —                                    | \( T(r), OX \leftarrow 1, FI \leftarrow 1, FR \leftarrow 0, XX \leftarrow 1, TX \) |
| Overflow     | Yes \(^1\)                          | No   | —    | Yes  | —                                    | —                                    | —                                    | \( T(r), OX \leftarrow 1, FI \leftarrow 1, FR \leftarrow 1, XX \leftarrow 1, TX \) |
| Overflow     | Yes \(^1\)                          | Yes  | —    | —    | —                                    | No                                   | No \(^2\)                            | \( Tw(q\beta), OX \leftarrow 1, FI \leftarrow 0, FR \leftarrow 0, TO \) |
| Overflow     | Yes \(^1\)                          | Yes  | —    | —    | Yes                                  | —                                    | —                                    | \( Tw(q\beta), OX \leftarrow 1, FI \leftarrow 1, FR \leftarrow 0, XX \leftarrow 1, TO \) |
| Overflow     | Yes \(^1\)                          | Yes  | —    | —    | Yes                                  | Yes                                  | Yes                                 | \( Tw(q\beta), OX \leftarrow 1, FI \leftarrow 1, FR \leftarrow 1, XX \leftarrow 1, TO \) |
| Normal       | No                                  | —    | —    | —    | —                                    | —                                    | —                                    | \( T(r), FI \leftarrow 0, FR \leftarrow 0 \) |
| Normal       | Yes                                 | —    | —    | No   | No                                   | —                                    | —                                    | \( T(r), FI \leftarrow 1, FR \leftarrow 0, XX \leftarrow 1 \) |
| Normal       | Yes                                 | —    | —    | Yes  | —                                    | —                                    | —                                    | \( T(r), FI \leftarrow 1, FR \leftarrow 1, XX \leftarrow 1 \) |
| Normal       | Yes                                 | —    | —    | Yes  | —                                    | —                                    | —                                    | \( T(r), FI \leftarrow 1, FR \leftarrow 0, XX \leftarrow 1, TX \) |
| Tiny         | No                                  | —    | —    | —    | —                                    | —                                    | —                                    | \( T(r), FI \leftarrow 0, FR \leftarrow 0 \) |
| Tiny         | No                                  | Yes  | —    | —    | —                                    | No \(^3\)                            | No \(^1\)                            | \( Tw(q\phi), UX \leftarrow 1, FI \leftarrow 0, FR \leftarrow 0, TU \) |
| Tiny         | Yes                                 | No   | No   | No   | —                                    | —                                    | —                                    | \( Tw(q\phi), UX \leftarrow 1, FI \leftarrow 1, FR \leftarrow 0, XX \leftarrow 1 \) |
| Tiny         | Yes                                 | No   | No   | Yes  | —                                    | —                                    | —                                    | \( Tw(q\phi), UX \leftarrow 1, FI \leftarrow 1, FR \leftarrow 1, XX \leftarrow 1 \) |
| Tiny         | Yes                                 | No   | No   | Yes  | —                                    | —                                    | —                                    | \( Tw(q\phi), UX \leftarrow 1, FI \leftarrow 1, FR \leftarrow 0, XX \leftarrow 1, TX \) |
| Tiny         | Yes                                 | No   | No   | Yes  | —                                    | —                                    | —                                    | \( Tw(q\phi), UX \leftarrow 1, FI \leftarrow 1, FR \leftarrow 1, XX \leftarrow 1 \) |
| Tiny         | Yes                                 | Yes  | —    | —    | No                                   | No \(^3\)                            | No \(^1\)                            | \( Tw(q\phi), UX \leftarrow 1, FI \leftarrow 0, FR \leftarrow 0, TU \) |
| Tiny         | Yes                                 | Yes  | —    | —    | Yes                                  | —                                    | —                                    | \( Tw(q\phi), UX \leftarrow 1, FI \leftarrow 1, FR \leftarrow 0, XX \leftarrow 1, TU \) |
| Tiny         | Yes                                 | Yes  | —    | —    | Yes                                  | Yes                                  | Yes                                 | \( Tw(q\phi), UX \leftarrow 1, FI \leftarrow 1, FR \leftarrow 1, XX \leftarrow 1, TU \) |

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.
- \(^\ast\) Rounding sets only the FI and FR status flags. Setting of the OX, XX, or UX flag is part of the exception actions. They are listed here for reference.
- \(\beta\) 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: \(\beta = 10^6\), where \(\alpha\) 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: \(\beta = 10^6\) where \(\kappa\) 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, FPSCR\(_{FI}\). This status flag is non-sticky.
- FR Floating-Point-Fraction-Rounded status flag, FPSCR\(_{FR}\).
- OX Floating-Point-Overflow Exception status flag, FPSCR\(_{OX}\).
- TO The system floating-point enabled exception error handler is invoked for the overflow exception if the FE0 and FE1 bits in the machine-state register 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 the FE0 and FE1 bits in the machine-state register 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 the FE0 and FE1 bits in the machine-state register 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 flag, FPSCR\(_{UX}\).
- XX Float-Point-Inexact-Exception Status flag, FPSCR\(_{XX}\). The flag is a sticky version of FPSCR\(_{FI}\). When FPSCR\(_{FI}\) is set to a new value, the new value of FPSCR\(_{XX}\) is set to the result of ORing the old value of FPSCR\(_{XX}\) with the new value of FPSCR\(_{FI}\).

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, FPXp denotes a FPR pair and must address an even-odd pair. If the FPXp field specifies an odd-numbered register, then the instruction form is invalid. The notation FPX[p] means either a FPR, FPX, or a FPR pair, FPXp.

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 FPSCR_{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 [Quad] X-form

<table>
<thead>
<tr>
<th>Instruction</th>
<th>FPRs</th>
<th>X-form</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>dadd</td>
<td>FRT,FRA,FRB</td>
<td>(Rc=0)</td>
<td></td>
</tr>
<tr>
<td>dadd.</td>
<td>FRT,FRA,FRB</td>
<td>(Rc=1)</td>
<td></td>
</tr>
<tr>
<td>daddq</td>
<td>FRT,FRA,FRBp</td>
<td>(Rc=0)</td>
<td></td>
</tr>
<tr>
<td>daddq.</td>
<td>FRT,FRA,FRBp</td>
<td>(Rc=1)</td>
<td></td>
</tr>
</tbody>
</table>

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 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 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 the FPSCR_{FPRF} field. The FPSCR_{FPRF} field 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.

**Special Registers Altered:**

<table>
<thead>
<tr>
<th>Field</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPRF</td>
<td>FR</td>
</tr>
<tr>
<td>FX</td>
<td>OX</td>
</tr>
<tr>
<td>VXSNAN</td>
<td>VXISI</td>
</tr>
</tbody>
</table>

(if Rc=1)

### DFP Subtract [Quad] X-form

<table>
<thead>
<tr>
<th>Instruction</th>
<th>FPRs</th>
<th>X-form</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>dsub</td>
<td>FRT,FRA,FRB</td>
<td>(Rc=0)</td>
<td></td>
</tr>
<tr>
<td>dsub.</td>
<td>FRT,FRA,FRB</td>
<td>(Rc=1)</td>
<td></td>
</tr>
<tr>
<td>dsubq</td>
<td>FRT,FRA,FRBp</td>
<td>(Rc=0)</td>
<td></td>
</tr>
<tr>
<td>dsubq.</td>
<td>FRT,FRA,FRBp</td>
<td>(Rc=1)</td>
<td></td>
</tr>
</tbody>
</table>

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 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 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 the FPSCR_{FPRF} field. The FPSCR_{FPRF} field 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.

**Special Registers Altered:**

<table>
<thead>
<tr>
<th>Field</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPRF</td>
<td>FR</td>
</tr>
<tr>
<td>FX</td>
<td>OX</td>
</tr>
<tr>
<td>VXSNAN</td>
<td>VXISI</td>
</tr>
</tbody>
</table>

(if Rc=1)
### Actions for Add (a + b) when operand b in FRB[p] is

<table>
<thead>
<tr>
<th>Operand a in FRA[p] is</th>
<th>-∞</th>
<th>F</th>
<th>+∞</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>-∞</td>
<td>T(-dINF)</td>
<td>T(-dINF)</td>
<td>V_{XISI}</td>
<td>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>
<td>(V_{XSNAN}: U(b))</td>
</tr>
<tr>
<td>+∞</td>
<td>V_{XISI}</td>
<td>T(dNaN)</td>
<td>T(+dINF)</td>
<td>P(b)</td>
<td>(V_{XSNAN}: U(b))</td>
</tr>
<tr>
<td>QNaN</td>
<td>P(a)</td>
<td>P(a)</td>
<td>P(a)</td>
<td>P(a)</td>
<td>(V_{XSNAN}: U(b))</td>
</tr>
<tr>
<td>SNaN</td>
<td>(V_{XSNAN}: U(a))</td>
<td>(V_{XSNAN}: U(a))</td>
<td>(V_{XSNAN}: U(a))</td>
<td>(V_{XSNAN}: U(a))</td>
<td>(V_{XSNAN}: 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 191)
- **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].
- **V_{XISI}**: The Invalid-Operation Exception (VXISI) occurs. The result is produced only when the exception is disabled. (See Section 5.5.10.1 “Invalid Operation Exception” on page 187 for the exception actions.)
- **V_{XSNAN}**: The Invalid-Operation Exception (VXSNAN) occurs. The result is produced only when the exception is disabled. (See Section 5.5.10.1 “Invalid Operation Exception” on page 187 for the exception actions.)

**Figure 78. Actions: Add**
**DFP Multiply [Quad] X-form**

<table>
<thead>
<tr>
<th>Dmul</th>
<th>FRT, FRA, FRB</th>
<th>(Rc=0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dmul.</td>
<td>FRT, FRA, FRB</td>
<td>(Rc=1)</td>
</tr>
</tbody>
</table>

The DFP operand in FRA[p] is multiplied by the DFP operand in 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 FRT[p]. The ideal exponent is the sum of the two exponents of the source operands.

Figure 79 summarizes the actions for Multiply. Figure 79 does not include the setting of the FPSCR_FPRF field. The FPSCR_FPRF field 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.

**Special Registers Altered:**

<table>
<thead>
<tr>
<th>FPRF FR FI FX OX UX XX VXSNAN VXIMZ CR1</th>
</tr>
</thead>
</table>

(if Rc=1)

---

**Operand a in FRA[p] is**

<table>
<thead>
<tr>
<th>Actions for Multiply (a*b) when operand b in FRB[p] is</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>------------</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>Fn</td>
</tr>
<tr>
<td>(\infty)</td>
</tr>
<tr>
<td>QNaN</td>
</tr>
<tr>
<td>SNaN</td>
</tr>
</tbody>
</table>

**Explanation:**

- **a * b**: The value a multiplied by b, rounded to the target-format precision and returned in the appropriate form. (See Section 5.5.11 on page 191)
- **dINF**: Default infinity.
- **dNaN**: Default quiet NaN.
- **Fn**: Finite nonzero number (includes both normal and subnormal numbers).
- **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 to the exclusive-OR of the source-operand signs.
- **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].
- **\(V_{\text{XIMZ}}\)**: The Invalid-Operation Exception (VXIMZ) occurs. The result is produced only when the exception is disabled. (See Section 5.5.10.1 “Invalid Operation Exception” on page 187 for the exception actions.)
- **\(V_{\text{XSNAN}}\)**: The Invalid-Operation Exception (VXSNAN) occurs. The result is produced only when the exception is disabled. (See Section 5.5.10.1 “Invalid Operation Exception” on page 187 for the exception actions.)

**Figure 79. Actions: Multiply**
DFP Divide [Quad]  X-form

ddiv  FRT,FRA,FRB (Rc=0)
ddiv. FRT,FRA,FRB (Rc=1)

ddivq FRTp,FRAp,FRBp (Rc=0)
ddivq. FRTp,FRAp,FRBp (Rc=1)

The DFP operand in FRA[p] is divided by the DFP operand in 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 FRT[p]. The ideal exponent is the difference of subtracting the exponent of the divisor from the exponent of the dividend.

Figure 80 summarizes the actions for Divide. Figure 80 does not include the setting of the FPSCR_FPRF field. The FPSCR_FPRF field is always set to the class and sign of the result, except for an enabled invalid-operation and enabled zero-divide exceptions, in which cases the field remains unchanged.

Special Registers Altered:

FPRF  FR  FI
FX  OX  UX  Zx  XX
VXSNAN  VXIDI  VXZDZ
CR1 (if Rc=1)

<table>
<thead>
<tr>
<th>63</th>
<th>FRTp</th>
<th>FRAp</th>
<th>FRBp</th>
<th>546</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>

Operand a in FRA[p] is

Actions for Divide (a ÷ b) when operand b in FRB[p] is

<table>
<thead>
<tr>
<th>0</th>
<th>Fn</th>
<th>∞</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>VXZDZ: T(dNaN)</td>
<td>S(a ÷ b)</td>
<td>S(zt)</td>
<td>P(b)</td>
<td>VXSNAN: U(b)</td>
</tr>
<tr>
<td>Fx: S(dINF)</td>
<td>S(a ÷ b)</td>
<td>S(zt)</td>
<td>P(b)</td>
<td>VXSNAN: U(b)</td>
</tr>
<tr>
<td>∞</td>
<td>S(dINF)</td>
<td>S(dINF)</td>
<td>VXIDI: T(dNaN)</td>
<td>P(b)</td>
</tr>
<tr>
<td>QNaN</td>
<td>P(a)</td>
<td>P(a)</td>
<td>P(a)</td>
<td>VXSNAN: U(b)</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 divided by b, rounded to the target-format precision and returned in the appropriate form. (See Section 5.5.11 on page 191.)
− dINF Default infinity.
− dNaN Default quiet NaN.
− Fn Finite nonzero number (includes both normal and subnormal numbers).
− 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 to the exclusive-OR of the source-operand signs.
− 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].
− VXIDI: The Invalid-Operation Exception (VXIDI) occurs. The result is produced only when the exception is disabled. (See Section 5.5.10.1 “Invalid Operation Exception” on page 187 for the exception actions.)
− VXSNAN: The Invalid-Operation Exception (VXSNAN) occurs. The result is produced only when the exception is disabled. (See Section 5.5.10.1 “Invalid Operation Exception” on page 187 for the exception actions.)
− VXZDZ: The Invalid-Operation Exception (VXZDZ) occurs. The result is produced only when the exception is disabled. (See Section 5.5.10.1 “Invalid Operation Exception” on page 187 for the exception actions.)
− 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 188 for the exception actions.)
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. The FPSCR_{FPCC} is set in the same way.

The codes in the CR field BF and FPSCR_{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 (FRA[p]) &lt; (FRB[p])</td>
</tr>
<tr>
<td>1</td>
<td>FG (FRA[p]) &gt; (FRB[p])</td>
</tr>
<tr>
<td>2</td>
<td>FE (FRA[p]) = (FRB[p])</td>
</tr>
<tr>
<td>3</td>
<td>FU (FRA[p]) ? (FRB[p])</td>
</tr>
</tbody>
</table>
**DFP Compare Unordered [Quad] X-form**

dcmu 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>6</td>
<td>9</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

dcmuq 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>6</td>
<td>9</td>
<td>16</td>
<td>21</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 FPSCRFPCC:

**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>
<th>-∞</th>
<th>F</th>
<th>+∞</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>-∞</td>
<td>AeqB</td>
<td>AltB</td>
<td>AltB</td>
<td>AuoB</td>
<td>Fu, VXSNAN</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>AgtB</td>
<td>C(a:b)</td>
<td>AltB</td>
<td>AuoB</td>
<td>Fu, VXSNAN</td>
<td></td>
</tr>
<tr>
<td>+∞</td>
<td>AgtB</td>
<td>AgtB</td>
<td>AeqB</td>
<td>AuoB</td>
<td>Fu, VXSNAN</td>
<td></td>
</tr>
<tr>
<td>NaN</td>
<td>AuoB</td>
<td>AuoB</td>
<td>AuoB</td>
<td>AuoB</td>
<td>Fu, VXSNAN</td>
<td></td>
</tr>
<tr>
<td>NaN</td>
<td>Fu, VXSNAN</td>
<td>Fu, VXSNAN</td>
<td>Fu, VXSNAN</td>
<td>Fu, VXSNAN</td>
<td>Fu, VXSNAN</td>
<td></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 FPSCRFPCC are set to 0b0010.
- AgtB CR field BF and FPSCRFPCC are set to 0b0100.
- AltB CR field BF and FPSCRFPCC are set to 0b1000.
- AuoB CR field BF and FPSCRFPCC are set to 0b0001.
- VXSNAN The invalid-operation exception (VXSNAN) occurs. See Section 5.5.10.1 for actions.

<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 [Quad] X-form

**dcmpo** BF,FRA,FRB

<table>
<thead>
<tr>
<th>59</th>
<th>BF // FRA</th>
<th>FRB</th>
<th>130</th>
<th>/</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>31</td>
</tr>
</tbody>
</table>

**dcmpoq** BF,FRAp,FRBp

<table>
<thead>
<tr>
<th>63</th>
<th>BF // FRAp</th>
<th>FRBp</th>
<th>130</th>
<th>/</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>11</td>
<td>16</td>
<td>21</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 FPSCR_{FPCC}.

**Special Registers Altered:**
- CR field BF
- FPCC
- FX VXSNAN VXVC

<table>
<thead>
<tr>
<th>Operand a in FRA[p] is</th>
<th>Actions for Compare ordered (a:b) when operand b in FRB[p] is</th>
</tr>
</thead>
<tbody>
<tr>
<td>-∞</td>
<td>AeqB, AltB, AuoB, VXSV</td>
</tr>
<tr>
<td>∞</td>
<td>AeqB, AltB, AuoB, VXVC</td>
</tr>
<tr>
<td>F</td>
<td>VSNAN</td>
</tr>
<tr>
<td>NaN</td>
<td>AeqB, VXSV, VXVC</td>
</tr>
<tr>
<td>NaN</td>
<td>AeqB, VXSV, VXVC</td>
</tr>
<tr>
<td>NaN</td>
<td>AeqB, VXSV, VXVC</td>
</tr>
<tr>
<td>NaN</td>
<td>AeqB, VXSV, VXVC</td>
</tr>
<tr>
<td>NaN</td>
<td>AeqB, VXSV, VXVC</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 FPSCR\( _{FPCC} \) are set to 0b0000.
- \( AltB \) CR field BF and FPSCR\( _{FPCC} \) are set to 0b0010.
- \( AgtB \) CR field BF and FPSCR\( _{FPCC} \) are set to 0b0100.
- \( AuoB \) CR field BF and FPSCR\( _{FPCC} \) are set to 0b0100.
- \( V_{XSV} \) The invalid-operation exception (VXSNAN) occurs. Additionally, if the exception is disabled (FPSCR\( _{VE}=0 \)), then FPSCR\( _{VXVC} \) is also set to one. See Section 5.5.10.1 for actions.
- \( V_{XVC} \) The invalid-operation exception (VXVC) occurs. See Section 5.5.10.1 for actions.

<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 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 [Quad]**

<table>
<thead>
<tr>
<th>dtstdc</th>
<th>BF,FRA,DCM</th>
</tr>
</thead>
<tbody>
<tr>
<td>59 BF // 9 FRA 11 DCM 194 /</td>
<td></td>
</tr>
<tr>
<td>0 6 9 1 1 6 2 2 3 1</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>dtstdcq</th>
<th>BF,FRAp,DCM</th>
</tr>
</thead>
<tbody>
<tr>
<td>63 BF // 9 FRAp 11 DCM 194 /</td>
<td></td>
</tr>
<tr>
<td>0 6 9 1 1 6 2 2 3 1</td>
<td></td>
</tr>
</tbody>
</table>

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.

**DCM Bit** | **Data Class**
--- | ---
0 | Zero
1 | Subnormal
2 | Normal
3 | Infinity
4 | Quiet NaN
5 | Signaling NaN

CR field BF and FPSCR\textsubscript{FPCC} are set to indicate the sign of the DFP operand in FRA[p] and whether the data class of the DFP operand in FRA[p] matches any of the data classes specified by DCM.

**Field** | **Meaning**
--- | ---
0000 | Operand positive with no match
0010 | Operand positive with match
1000 | Operand negative with no match
1010 | Operand negative with match

**Special Registers Altered:**

CR field BF
FPCC

**DFP Test Data Group [Quad]**

<table>
<thead>
<tr>
<th>dtstdg</th>
<th>BF,FRA,DGM</th>
</tr>
</thead>
<tbody>
<tr>
<td>59 BF // 9 FRA 11 DGM 226 /</td>
<td></td>
</tr>
<tr>
<td>0 6 9 1 1 6 2 2 3 1</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>dtstdgq</th>
<th>BF,FRAp,DGM</th>
</tr>
</thead>
<tbody>
<tr>
<td>63 BF // 9 FRAp 11 DGM 226 /</td>
<td></td>
</tr>
<tr>
<td>0 6 9 1 1 6 2 2 3 1</td>
<td></td>
</tr>
</tbody>
</table>

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, $X_{\text{min}}$.

**DGM Bit** | **Data Group**
--- | ---
0 | Zero with non-extreme exponent
1 | Zero with extreme exponent
2 | Subnormal or (Normal with extreme exponent)
3 | Normal with non-extreme exponent and leftmost zero digit in significand
4 | Normal with non-extreme exponent and leftmost nonzero digit in significand
5 | Special symbol (Infinity, QNaN, or SNaN)

CR field BF and FPSCR\textsubscript{FPCC} are set to indicate the sign of the DFP operand in FRA[p] and whether the data group of the DFP operand in FRA[p] matches any of the data groups specified by DGM.

**Field** | **Meaning**
--- | ---
0000 | Operand positive with no match
0010 | Operand positive with match
1000 | Operand negative with no match
1010 | Operand negative with match

**Special Registers Altered:**

CR field BF
FPCC
DFP Test Exponent [Quad] X-form

dstex BF,FRA,FRB

\[
\begin{array}{cccccc}
59 & BF & 6 & FRA & 9 & FRB & 16 & 162 & / \\
\end{array}
\]

dstexq BF,FRAp,FRBp

\[
\begin{array}{cccccc}
63 & BF & 6 & FRAp & 9 & FRBp & 16 & 162 & / \\
\end{array}
\]

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 FPSCRFPCC.

The codes in the CR field BF and FPSCRFPCC are defined for the DFP Test Exponent operations as follows.

### Special Registers Altered:
- CR field BF
- FPCC

### Bit Description

<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>

### Explanation:
- \( C(Ea:Eb) \) Algebraic comparison. See the table below.
- F All finite numbers, including zeros
- AeqB CR field BF and FPSCRFPCC are set to 0b0010.
- AgtB CR field BF and FPSCRFPCC are set to 0b0100.
- AltB CR field BF and FPSCRFPCC are set to 0b1000.
- AuoB CR field BF and FPSCRFPCC 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
Let \( k \) be the contents of bits 58:63 of \( FPR_{[FRA]} \) that specifies the reference significance.

For \( dtstsf \), let the value \( NSD_{db} \) be the number of significant digits of the DFP value in \( FPR_{[FRB]} \).

For \( dtstsfq \), let the value \( NSD_{db} \) be the number of significant digits of the DFP value in \( FPR_{[FRB_{p} : FRB_{p+1}]} \).

For this instruction, the number of significant digits of the value 0 is considered to be zero.

\( NSD_{db} \) is compared to \( k \). The result of the compare is placed into CR field \( BF \) and the FPCC as follows.

Special Registers Altered:
- CR field \( BF \)
- FPCC

Figure 84. Actions: Test Significance

**Programming Note**

The reference significance can be loaded into a FPR using a *Load Float as Integer Word Algebraic* instruction.
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 183 for the definition of RMC encoding.

All Quantum Adjustment instructions set the FI and FR status flags, and also set the FPSCR\textsubscript{FPRF} field. The record bit is provided to each of these instructions. They return the target operand in a form with the ideal exponent.

**DFP Quantize Immediate [Quad] Z23-form**

\[
\text{dquai} \quad \text{TE,FRT,FRB,RMC} \quad (\text{Rc}=0) \\
\text{dquai.} \quad \text{TE,FRT,FRB,RMC} \quad (\text{Rc}=1)
\]

<table>
<thead>
<tr>
<th>59</th>
<th>FRT</th>
<th>TE</th>
<th>FRB</th>
<th>RMC</th>
<th>67</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>
</tr>
</tbody>
</table>

\[
\text{dquaq} \quad \text{TE,FRTp,FRBp,RMC} \quad (\text{Rc}=0) \\
\text{dquaq.} \quad \text{TE,FRTp,FRBp,RMC} \quad (\text{Rc}=1)
\]

<table>
<thead>
<tr>
<th>63</th>
<th>FRTp</th>
<th>TE</th>
<th>FRBp</th>
<th>RMC</th>
<th>67</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>
</tr>
</tbody>
</table>

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 \((10^p-1)\%10^{1E}\), 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].

The FPSCR\textsubscript{FPRF} field 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.

**Special Registers Altered:**

\[
\begin{align*}
\text{FPRF} & \quad \text{FR} & \quad \text{FI} \\
\text{FX} & \quad \text{XX} \\
\text{VXSNAN} & \quad \text{VXCVI} \\
\text{CR1} & \quad \text{(if Rc=1)}
\end{align*}
\]

**Programming Note**

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 * FRB) to 2 fractional digits (TE field = -2) using Round to nearest, ties away from 0 (RMC field = 2). The quantized and rounded result is placed in FRT.

\[
dmul \quad f0,\text{FRA},\text{FRB} \\
dqua -2,\text{FRT},f0,2
\]
**DFP Quantize [Quad] Z23-form**

dqua  | FRT,FRA,FRB,RMC  
--- | ---  
   | (Rc=0)  
dqua. | FRT,FRA,FRB,RMC  
--- | ---  
   | (Rc=1)  

| dquaq  | FRTp,FRAp,FRBp,RMC  
--- | ---  
   | (Rc=0)  
dquaq. | FRTp,FRAp,FRBp,RMC  
--- | ---  
   | (Rc=1)  

The DFP operand in register FRB[p] is converted and rounded to the form with the same exponent as that of the DFP operand in FRA[p] based on the rounding mode specified in the RMC field. 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 in FRA[p].

When the value of the operand in FRB[p] is greater than \((10^p - 1) \% 10^{Ea}\), where \(p\) is the format precision and \(Ea\) is the exponent of the operand in FRA[p], 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].

Figure 87 and Figure 88 summarize the actions. The tables do not include the setting of the FPSCHR field. The FPSCHR field 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.

**Special Register Altered:**

<table>
<thead>
<tr>
<th>FPRF</th>
<th>FR</th>
<th>FI</th>
</tr>
</thead>
<tbody>
<tr>
<td>FX</td>
<td>XX</td>
<td></td>
</tr>
<tr>
<td>VXSNAN</td>
<td>VXCVI</td>
<td></td>
</tr>
<tr>
<td>CR1</td>
<td></td>
<td>(if Rc=1)</td>
</tr>
</tbody>
</table>

**Programming Note**

*DFP Quantize* can be used to adjust one DFP value (FRB[p]) to a form having the same exponent as a second DFP value (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 the RMC field. Figure 86 shows examples of these adjustments.

<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 10^0) )</td>
<td>9 ( (9 \times 10^0) )</td>
<td>9 ( (9 \times 10^0) )</td>
</tr>
<tr>
<td>1.00 ( (100 \times 10^{-2}) )</td>
<td>9. ( (9 \times 10^0) )</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 ( (9 \times 10^0) )</td>
<td>49 ( (9 \times 10^0) )</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>49 ( (9 \times 10^0) )</td>
<td>50 ( (5 \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>0.001 ( (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^{-1}) )</td>
<td>9999999999999999 ( (9999999999999999 \times 10^0) )</td>
<td>QNaN</td>
<td>QNaN</td>
</tr>
</tbody>
</table>

Figure 86. DFP Quantize examples
### Chapter 5. Decimal Floating-Point

#### Figure 87. Actions (part 1) Quantize

<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>(\infty)</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>*</td>
<td>*</td>
<td>V_{XCVI}: T(dNaN)</td>
<td>P(b)</td>
<td>V_{XSNAN}: U(b)</td>
</tr>
<tr>
<td>Fn</td>
<td></td>
<td>*</td>
<td>*</td>
<td>V_{XCVI}: T(dNaN)</td>
<td>P(b)</td>
<td>V_{XSNAN}: U(b)</td>
</tr>
<tr>
<td>*</td>
<td></td>
<td>V_{XCVI}: T(dNaN)</td>
<td>V_{XCVI}: T(dNaN)</td>
<td>T(dINF)</td>
<td>P(b)</td>
<td>V_{XSNAN}: U(b)</td>
</tr>
<tr>
<td>QNaN</td>
<td>P(a)</td>
<td>P(a)</td>
<td>P(a)</td>
<td>P(a)</td>
<td>V_{XSNAN}: U(a)</td>
<td></td>
</tr>
<tr>
<td>SNaN</td>
<td>V_{XSNAN}: U(a)</td>
<td>V_{XSNAN}: U(a)</td>
<td>V_{XSNAN}: U(a)</td>
<td>V_{XSNAN}: U(a)</td>
<td></td>
<td></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].
- V_{XCVI} The Invalid-Operation Exception (VXCVI) occurs. The result is produced only when the exception is disabled. (See Section 5.5.10.1 for actions)
- V_{XSNAN} The Invalid-Operation Exception (VXSNAN) occurs. The result is produced only when the exception is disabled. (See Section 5.5.10.1 for actions)

#### Figure 88. Actions (part 2) Quantize

<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>(10^p - 1) % 10^{16}</td>
<td>V_{b} &gt; (10^p - 1) % 10^{16}</td>
<td>E(0)</td>
<td>V_{XCVI}: T(dNaN)</td>
</tr>
<tr>
<td>(10^p - 1) % 10^{16}</td>
<td>V_{b} \leq (10^p - 1) % 10^{16}</td>
<td>E(0)</td>
<td>L(b)</td>
</tr>
<tr>
<td>Te = Se</td>
<td>E(0)</td>
<td>W(b)</td>
<td></td>
</tr>
<tr>
<td>Te &gt; Se</td>
<td>E(0)</td>
<td>QR(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 FBR[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].
- V_{b} The value of the operand in FRB[p].
- W(x) The value and the form of operand x is placed in FRT[p].
- V_{XCVI} The Invalid-Operation Exception (VXCVI) occurs. The result is produced only when the exception is disabled. (See Section 5.5.10.1 for actions.)
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 in the RMC field.

If the exponent of the rounded result of the form that has the specified number of significant digits would be greater than X\text{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 the FPSCR\text{FPRF} field. The FPSCR\text{FPRF} field 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.

**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 the RMC field. 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 combined with DFP Quantize* can be used to left justify a value (as needed by the frexp function). FRB is the DFP value for which we want to left justify; f13 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.

```assembly
  drrnd f1,f13,FRB,1 # reround 1 digit toward 0
  dxex f0,f1
  stfd f0,-8(r1)
  lfd r11,-8(r1)
  addi r11,r11,15 # biased exp + precision - 1
  lfd r11,-8(r1)
  stfd f0,-8(r1)
  diex f1,f0,f1 # adjust exponent
  dqua FRT,f1,f0,1 # quantize to adjusted exponent
```
### Actions for Reround when operand b in FRB[p] is

<table>
<thead>
<tr>
<th></th>
<th>0*</th>
<th>Fn</th>
<th>∞</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>k ≥ 0, k &lt; m</td>
<td>-</td>
<td>RR(b) or V_XCVI: T(dNaN)</td>
<td>T(dINF)</td>
<td>P(b)</td>
<td>V_XSNAN: U(b)</td>
</tr>
<tr>
<td>k ≥ 0, k = m</td>
<td>-</td>
<td>W(b)</td>
<td>T(dINF)</td>
<td>P(b)</td>
<td>V_XSNAN: U(b)</td>
</tr>
<tr>
<td>k ≥ 0 and k &gt; m, or k = 0</td>
<td>W(b)</td>
<td>W(b)</td>
<td>T(dINF)</td>
<td>P(b)</td>
<td>V_XSNAN: U(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.
- Default infinity.
- Finite nonzero numbers (includes both subnormal and normal numbers).
- Reference significance, which specifies the number of significant digits in the target operand.
- Number of significant digits in the operand in FRB[p].
- The QNaN of operand x is propagated and placed in FRT[p].
- The value x is rounded to the form that has the specified number of significant digits. If RR(x) \( \left(10^k-1\right) \% 10^{\text{max}} \), then RR(x) is returned; otherwise an invalid-operation exception is recognized.
- The value x is placed in FRT[p].
- The SNaN of operand x is converted to the corresponding QNaN and placed in FRT[p].
- The Invalid-Operation Exception (V\_XCVI) occurs. The result is produced only when the exception is disabled. (See Section 5.5.10.1 for actions.)
- The Invalid-Operation Exception (V\_XSNAN) occurs. The result is produced only when the exception is disabled. See Section 5.5.10.1 for actions.
- The value and the form of x is placed in FRT[p].

**Figure 90. Actions: Reround**
DFP Round To FP Integer With Inexact [Quad] Z23-form

\[
\begin{array}{cccccc}
\text{drintx} & \text{FRT}, \text{FRB}, \text{RMC} & (\text{Rc}=0) \\
\text{drintx} & \text{FRT}, \text{FRB}, \text{RMC} & (\text{Rc}=1)
\end{array}
\]

\[
\begin{array}{cccccc}
59 & \text{FRT} & / / & R & \text{FRB} & \text{RMC} & 99 & \text{Rc} \\
0 & 6 & 11 & 15 & 16 & 21 & 23 & 31
\end{array}
\]

\[
\begin{array}{cccccc}
\text{drintxq} & \text{FRTp}, \text{FRBp}, \text{RMC} & (\text{Rc}=0) \\
\text{drintxq} & \text{FRTp}, \text{FRBp}, \text{RMC} & (\text{Rc}=1)
\end{array}
\]

\[
\begin{array}{cccccc}
63 & \text{FRTp} & / / & R & \text{FRBp} & \text{RMC} & 99 & \text{Rc} \\
0 & 6 & 11 & 15 & 16 & 21 & 23 & 31
\end{array}
\]

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 in the RMC field.
When the RMC-encoding-selection (R) bit is zero, the 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 the FPSCR_FPRF field. The FPSCR_FPRF field is always set to the class and sign of the result, except for an enabled invalid-operation, in which case the field remains unchanged.

**Special Registers Altered:**
- FP_FPR  FR  F1
- FX  XX
- VXSNAN
- CR1  (if Rc=1)

**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 FPSCR_DRN (R=0, RMC=11). The specification for rint requires the inexact exception be raised if detected.
### Figure 91. Actions: Round to FP Integer With Inexact

| Operand b in FRB is | Is n not precise (n ≠ b) | Inv.-Op. Exception Enabled | Inexact Exception Enabled | Is n Incremented (|n| > |b|) | Actions* |
|---------------------|--------------------------|-----------------------------|---------------------------|-----------------|----------|
| \(-\infty\)        | No\(^1\)                 | -                           | -                         | -               | \(-d\infty\), FI ← 0, FR ← 0 |
| F                   | No                       | -                           | -                         | -               | W(n), FI ← 0, FR ← 0 |
| F                   | Yes                      | -                           | No                        | No              | W(n), FI ← 1, FR ← 0, XX ← 1 |
| F                   | Yes                      | -                           | No                        | Yes             | W(n), FI ← 1, FR ← 1, XX ← 1 |
| F                   | Yes                      | -                           | Yes                       | No              | W(n), FI ← 1, FR ← 0, XX ← 1, TX |
| F                   | Yes                      | -                           | Yes                       | Yes             | W(n), FI ← 1, FR ← 1, XX ← 1, TX |
| +\(\infty\)        | No\(^1\)                 | -                           | -                         | -               | \(+d\infty\), FI ← 0, FR ← 0 |
| QNaN                | No\(^1\)                 | -                           | -                         | -               | P(b), FI ← 0, FR ← 0 |
| SNaN                | No\(^1\)                 | No                          | -                         | -               | U(b), FI ← 0, FR ← 0, VXSNAN ← 1 |
| SNaN                | No\(^1\)                 | Yes                         | -                         | -               | VXSNAN ← 1, TV |

**Explanation:**
- Setting of XX and VXSNAN is part of the corresponding exception actions. Also, when an invalid-operation exception occurs, setting of FI and 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.

\(d\infty\)  Default infinity.

F  All finite numbers, including zeros.

FI  Floating-Point-Fraction-Inexact status flag, FPSCR\(_{FI}\).

FR  Floating-Point-Fraction-Rounded status flag, FPSCR\(_{FR}\).

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 the FE0 and FE1 bits in the machine-state register 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 the FE0 and FE1 bits in the machine-state register 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 FPT\([p]\).

W(x)  The value x in the form of zero exponent or the source exponent is placed in FRT\([p]\).

XX  Floating-Point-Inexact-Exception status flag, FPSCR\(_{XX}\).
DFP Round To FP Integer Without Inexact

[Quad] \ Z23-form

\[
\begin{array}{cccccccc}
\text{drin}n & \text{drin}n. & \text{drint}n & \text{drint}n. \\
\text{R,FRT,FRB,RMC} & \text{R,FRT,FRB,RMC} & \text{R,FRT,FRB,RMC} & \text{R,FRT,FRB,RMC} \\
\text{(Rc=0)} & \text{(Rc=1)} & \text{(Rc=0)} & \text{(Rc=1)} \\
\hline
\text{59} & \text{6} & \text{11} & \text{15} & \text{16} & \text{21} & \text{23} & \text{227} & \text{Rc} \\
\end{array}
\]

Special Registers Altered:
- FPRF FR (set to 0) FI (set to 0)
- VXSNAN CR1

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.

Function | R | RMC
---|---|---
Ceil | 1 | 0b00
Floor | 1 | 0b01
Nearbyint | 0 | 0b11
Round | 0 | 0b10
Trunc | 0 | 0b01

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.

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 the FPRSCRFPRF field. The FPRSCRFPRF field is always set to the class and sign of the result, except for an enabled invalid-operation, in which case the field remains unchanged.

<table>
<thead>
<tr>
<th>Operand b in FRB is</th>
<th>Inv.-Op. Exception Enabled</th>
<th>Actions*</th>
</tr>
</thead>
<tbody>
<tr>
<td>-( \infty )</td>
<td>-</td>
<td>T(-dINF), FI ( \leftarrow 0 ), FR ( \leftarrow 0 )</td>
</tr>
<tr>
<td>( F )</td>
<td>-</td>
<td>W(n), FI ( \leftarrow 0 ), FR ( \leftarrow 0 )</td>
</tr>
<tr>
<td>+( \infty )</td>
<td>-</td>
<td>T(+dINF), FI ( \leftarrow 0 ), FR ( \leftarrow 0 )</td>
</tr>
<tr>
<td>QNaN</td>
<td>-</td>
<td>P(b), FI ( \leftarrow 0 ), FR ( \leftarrow 0 )</td>
</tr>
<tr>
<td>SNaN</td>
<td>No</td>
<td>U(b), FI ( \leftarrow 0 ), FR ( \leftarrow 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 FI 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.
- Stif default infinity.
- All finite numbers, including zeros.
- Floating-Point-Fraction-Inexact status flag, FPRSCRFi.
- Floating-Point-Fraction-Rounded status flag, FPRSCRFR.
- The value derived when the source operand, b, is rounded to an integer using the special rounding for Round-To-FP-Integer.
- The QNaN of operand x is propagated and placed in FRT[p].
- The value x is placed in FRT[p].
- The system floating-point enabled exception error handler is invoked for the invalid-operation exception if the FE0 and FE1 bits in the machine-state register are set to any mode other than the ignore-exception mode.
- The SNaN of operand x is converted to the corresponding QNaN and placed in FPT[p].
- 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>F</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. dINF Default infinity.
7. F All finite numbers, including zeros.
8. P(x) The special symbol in operand x is propagated into FRT[p].
9. R(x) The value x is rounded to the target-format precision; see Section 5.5.11
10. T(x) The value x is placed in FRT[p].
11. U(x) The SNaN of operand x is converted to the corresponding QNaN.
12. VXSNAN The Invalid-Operation Exception (VXSNAN) occurs. The result is produced only when the exception is disabled. See Section 5.5.10.1 for actions.

Figure 93. Actions: Data-Format Conversion Instructions
**DFP Convert To DFP Long**  

\[
dctdp \quad \text{FRT,FRB} \quad \text{(Rc=0)} \\
dctdp. \quad \text{FRT,FRB} \quad \text{(Rc=1)}
\]

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.

**Special Registers Altered:**
- FPRF: FR (undefined)  
  - FI (undefined)  
- CR1 (if 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 Convert To DFP Extended**  

\[
dctpq \quad \text{FRTp,FRB} \quad \text{(Rc=0)} \\
dctpq. \quad \text{FRTp,FRB} \quad \text{(Rc=1)}
\]

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.

**Special Registers Altered:**
- FPRF: FR (set to 0)  
  - FI (set to 0)  
- FX  
- VXSNAN  
- CR1 (if 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.

**Special Registers Altered:**
- FPRF FR FI
- FX OX UX XX
- CR1 (if Rc=1)

**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.

---

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.

**Special Registers Altered:**
- FPRF FR FI
- FX OX UX XX
- VXSNAN
- CR1 (if Rc=1)

**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.
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.

**DFP Convert From Fixed X-form**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dcffix</td>
<td>FRT,FRB</td>
</tr>
<tr>
<td>dcffix.</td>
<td>FRT,FRB</td>
</tr>
</tbody>
</table>

**X-form**

<table>
<thead>
<tr>
<th>59</th>
<th>FRT</th>
<th>///</th>
<th>FRB</th>
<th>802</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>

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.

The FPSCRFP field is set to the class and sign of the result.

**Special Registers Altered:**

- FPRF FR FI
- FX XX
- CR1

**DFP Convert From Fixed Quad X-form**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dcffixq</td>
<td>FRTp,FRB</td>
</tr>
<tr>
<td>dcffixq.</td>
<td>FRTp,FRB</td>
</tr>
</tbody>
</table>

**X-form**

<table>
<thead>
<tr>
<th>63</th>
<th>FRTp</th>
<th>///</th>
<th>FRB</th>
<th>802</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>

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.

The FPSCRFP field is set to the class and sign of the result.

**Special Registers Altered:**

- FPRF (undefined) FR FI
- FX XX
- VXSNAN VXCVI
- CR1

**DFP Convert To Fixed [Quad] X-form**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dctfixq</td>
<td>FRT,FRBp</td>
</tr>
<tr>
<td>dctfixq.</td>
<td>FRT,FRBp</td>
</tr>
</tbody>
</table>

**X-form**

<table>
<thead>
<tr>
<th>59</th>
<th>FRT</th>
<th>///</th>
<th>FRB</th>
<th>290</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>

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.

**Special Registers Altered:**

- FPRF 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]` is a rounding mode other than the current rounding mode specified by FPSCRDRN is needed. Saving, modifying and restoring the FPSCR just to temporarily change the rounding mode is less efficient than just employing drint[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 FPSCRDRN) is Round to Nearest, Ties to Even then following is preferred.

```
drintn 0,f1,f1,2
dctfix  f1,f1
```

Version 3.0 B
| Operand b in FRB[p] is | q is | Is n not precise (\(n \neq b\)) | Inv.-Op. Except. Enabled | Inexact Except. Enabled | Is n Incremented (\(|n| > |b|\)) | Actions * |
|------------------------|------|---------------------------------|--------------------------|------------------------|-------------------------------|----------|
| \(-\infty \leq b < MN\) | \(-\infty \leq b < MN\) | - No | - | - | T(MN), FI \(\leftarrow 0\), FR \(\leftarrow 0\), VXCVI \(\leftarrow 1\) |
| MN \(\leq b < 0\) | MN \(\leq b < 0\) | - Yes | - No | - | T(MN), FI \(\leftarrow 1\), FR \(\leftarrow 0\), XX \(\leftarrow 1\), TX | |
| MN \(\leq b < 0\) | MN \(\leq b < 0\) | - Yes | - Yes | No | T(n), FI \(\leftarrow 1\), FR \(\leftarrow 0\), XX \(\leftarrow 1\) |
| MN \(\leq b < 0\) | MN \(\leq b < 0\) | - Yes | - Yes | Yes | T(n), FI \(\leftarrow 1\), FR \(\leftarrow 1\), XX \(\leftarrow 1\), TX |
| MN \(\leq b < 0\) | MN \(\leq b < 0\) | - Yes | - Yes | Yes | T(n), FI \(\leftarrow 1\), FR \(\leftarrow 1\), XX \(\leftarrow 1\), TX |
| 0 < b \(\leq MP\) | 0 < b \(\leq MP\) | - No | - | - | T(n), FI \(\leftarrow 0\), FR \(\leftarrow 0\) |
| 0 < b \(\leq MP\) | 0 < b \(\leq MP\) | - Yes | - No | No | T(n), FI \(\leftarrow 1\), FR \(\leftarrow 0\), XX \(\leftarrow 1\) |
| 0 < b \(\leq MP\) | 0 < b \(\leq MP\) | - Yes | - No | Yes | T(n), FI \(\leftarrow 1\), FR \(\leftarrow 1\), XX \(\leftarrow 1\) |
| 0 < b \(\leq MP\) | 0 < b \(\leq MP\) | - Yes | - Yes | No | T(n), FI \(\leftarrow 1\), FR \(\leftarrow 0\), XX \(\leftarrow 1\), TX |
| 0 < b \(\leq MP\) | 0 < b \(\leq MP\) | - Yes | - Yes | Yes | T(n), FI \(\leftarrow 1\), FR \(\leftarrow 1\), XX \(\leftarrow 1\), TX |
| MP < b \(\leq +\infty\) | MP < b \(\leq +\infty\) | - No | - | - | T(MP), FI \(\leftarrow 1\), FR \(\leftarrow 0\), XX \(\leftarrow 1\) |
| MP < b \(\leq +\infty\) | MP < b \(\leq +\infty\) | - Yes | - Yes | - | T(MP), FI \(\leftarrow 1\), FR \(\leftarrow 0\), VXCVI \(\leftarrow 1\), TV |
| QNaN | - | - No | - | - | T(MN), FI \(\leftarrow 0\), FR \(\leftarrow 0\), VXCVI \(\leftarrow 1\) |
| SNaN | - | - No | - | - | VXCVI \(\leftarrow 1\), VXCVI \(\leftarrow 1\), VXSNAN \(\leftarrow 1\) |
| SNaN | - | - Yes | - | - | VXCVI \(\leftarrow 1\), VXCVI \(\leftarrow 1\), VXSNAN \(\leftarrow 1\), TX |

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 flag, FPSCR\(_{FI}\).
FR Floating-Point-Fraction-Rounded status flag, FPSCR\(_{FR}\).
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 the FEO and FE1 bits in the machine-state register 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 the FEO and FE1 bits in the machine-state register are set to any mode other than the ignore-exception mode.
VXCVI The FPSCR\(_{VXCVI}\) invalid operation exception status bit.
VXSNAN The FPSCR\(_{VXSNAN}\) invalid operation exception status bit.
XX Floating-Point-Inexact-Exception status flag, FPSCR\(_{XX}\).
### 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).

The format instructions consist of Decode DPD To BCD, Encode BCD To DPD, Extract Biased Exponent, Insert Biased Exponent, Shift Significand Left Immediate, and Shift Significand Right Immediate.

#### DFP Decode DPD To BCD [Quad] 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>
<tr>
<td>59 6 11 13 16 21</td>
<td>322</td>
<td>Rc 31</td>
</tr>
</tbody>
</table>

<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>
<tr>
<td>63 6 11 13 16 21</td>
<td>322</td>
<td>Rc 31</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).

**Special Registers Altered:**

CR1 (if Rc=1)

#### DFP Encode BCD To DPD [Quad] 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>
<tr>
<td>59 6 11 12 16 21</td>
<td>834</td>
<td>Rc 31</td>
</tr>
</tbody>
</table>

<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>
<tr>
<td>63 6 11 12 16 21</td>
<td>834</td>
<td>Rc 31</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.

FPSCR_FPR is set to the class and sign of the result, except for Invalid Operation Exception when FPSCR_VE=1.

**Special Registers Altered:**

FPRF FR (set to 0) FI (set to 0) FX VXCVI CR1 (if Rc=1)
**DFP Extract Biased Exponent [Quad] X-form**

<table>
<thead>
<tr>
<th>dxex</th>
<th>FRT,FRB</th>
<th>(Rc=0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>dxex.</td>
<td>FRT,FRB</td>
<td>(Rc=1)</td>
</tr>
</tbody>
</table>

The biased exponent of the operand in FRB[p] is extracted and placed into FRT in the 64-bit signed binary integer format. When the operand in FRB is an infinity, QNaN, or SNaN, a special code is returned.

**Operand**  
Finite Number: biased exponent value  
Infinity: -1  
QNaN: -2  
SNaN: -3

**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 [Quad] X-form**

<table>
<thead>
<tr>
<th>dxex</th>
<th>FRT,FRA,FRB</th>
<th>(Rc=0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>dxex.</td>
<td>FRT,FRA,FRB</td>
<td>(Rc=1)</td>
</tr>
</tbody>
</table>

Let a be the value of the 64-bit signed binary integer in FRA.

### Result

- $a > MBE^1$: QNaN
- $a = -1$: Infinity
- $a = -2$: QNaN
- $a = -3$: SNaN
- $a < -3$: QNaN

$^1$ Maximum biased exponent for the target format

When $0 \leq a \leq MBE$, a is the biased target exponent that is combined with the sign bit and the significand value of the DFP operand in FRB[p] to form the DFP result in FRT[p]. The ideal exponent is the specified target exponent.

When a specifies a special code ($a < 0$ or $a > MBE$), an infinity, QNaN, or SNaN is formed in FRT[p] with the trailing significand field containing the value from the trailing significand field of the source operand in 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.

**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

| Actions for Insert Biased Exponent when operand b in FRB[p] specifies |
|--------------------------|------------------|-----------------|------------------|
| F                        | ∞                | QNaN            | SNaN             |
| F                        | N, Rb            | Z, Rb           | Z, Rb            |
| QNaN                     | Z, Rb            | Z, Rb           | Z, Rb            |
| SNaN                     | Z, Rb            | Z, Rb           | Z, Rb            |

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 [Quad] Z22-form**

<table>
<thead>
<tr>
<th>59</th>
<th>FRT</th>
<th>FRA</th>
<th>SH</th>
<th>66</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>22</td>
<td>31</td>
</tr>
</tbody>
</table>

dscli FRT,FRA,SH (Rc=0)
dscli. FRT,FRA,SH (Rc=1)
dscliq FRTp,FRAp,SH (Rc=0)
dscliq. FRTp,FRAp,SH (Rc=1)

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.

**Special Registers Altered:**
CR1 (if Rc=1)

---

**DFP Shift Significand Right Immediate [Quad] Z22-form**

<table>
<thead>
<tr>
<th>59</th>
<th>FRT</th>
<th>FRA</th>
<th>SH</th>
<th>98</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>22</td>
<td>31</td>
</tr>
</tbody>
</table>

dscri FRT,FRA,SH (Rc=0)
dscri. FRT,FRA,SH (Rc=1)
dscriq FRTp,FRAp,SH (Rc=0)
dscriq. FRTp,FRAp,SH (Rc=1)

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.

**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>Vs</th>
<th>G</th>
<th>Encoding</th>
<th>FPRF</th>
<th>FPCC</th>
<th>FP Exception</th>
<th>FRFI</th>
<th>IE</th>
<th>( \infty )</th>
</tr>
</thead>
<tbody>
<tr>
<td>dadd</td>
<td>DFP Add</td>
<td>X</td>
<td>FRT, FRA, FRB</td>
<td>Y</td>
<td>N</td>
<td>RE</td>
<td>Y Y</td>
<td>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</td>
<td>N</td>
<td>RE</td>
<td>Y Y</td>
<td>V</td>
<td>O U X</td>
<td>Y Y Y</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dsbb</td>
<td>DFP Subtract</td>
<td>X</td>
<td>FRT, FRA, FRB</td>
<td>Y</td>
<td>N</td>
<td>RE</td>
<td>Y Y V</td>
<td>O U X</td>
<td>Y Y Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dsbbq</td>
<td>DFP Subtract Quad</td>
<td>X</td>
<td>FRTp, FRAp, FRBp</td>
<td>Y</td>
<td>N</td>
<td>RE</td>
<td>Y Y V</td>
<td>O U X</td>
<td>Y Y Y</td>
<td></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</td>
<td>N</td>
<td>RE</td>
<td>Y Y</td>
<td>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</td>
<td>N</td>
<td>RE</td>
<td>Y Y V</td>
<td>O U X</td>
<td>Y Y Y</td>
<td></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</td>
<td>N</td>
<td>RE</td>
<td>Y Y</td>
<td>V Z</td>
<td>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</td>
<td>N</td>
<td>RE</td>
<td>Y Y V</td>
<td>Z O</td>
<td>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</td>
<td>Y V</td>
<td>- -</td>
<td>N</td>
<td></td>
<td></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</td>
<td>Y V</td>
<td>- -</td>
<td>N</td>
<td></td>
<td></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</td>
<td>Y V</td>
<td>- -</td>
<td>N</td>
<td></td>
<td></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</td>
<td>Y V</td>
<td>- -</td>
<td>N</td>
<td></td>
<td></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</td>
<td>Y</td>
<td>- -</td>
<td>N</td>
<td></td>
<td></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</td>
<td>Y</td>
<td>- -</td>
<td>N</td>
<td></td>
<td></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</td>
<td>Y</td>
<td>- -</td>
<td>N</td>
<td></td>
<td></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</td>
<td>Y</td>
<td>- -</td>
<td>N</td>
<td></td>
<td></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</td>
<td>Y</td>
<td>- -</td>
<td>N</td>
<td></td>
<td></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</td>
<td>Y</td>
<td>- -</td>
<td>N</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dtstsf</td>
<td>DFP Test Significance</td>
<td>X</td>
<td>BF, FRAP(FIX), FRB</td>
<td>N - -</td>
<td>N</td>
<td>Y</td>
<td>- -</td>
<td>N</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dtstsfq</td>
<td>DFP Test Significance Quad</td>
<td>X</td>
<td>BF, FRAP(FIX), FRBp</td>
<td>N - -</td>
<td>N</td>
<td>Y</td>
<td>- -</td>
<td>N</td>
<td></td>
<td></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</td>
<td>RE</td>
<td>Y Y V</td>
<td>X Y Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dquaiq</td>
<td>DFP Quantize Immediate Quad</td>
<td>Z23</td>
<td>TEP, FRTP, FRBP, RMC</td>
<td>Y N</td>
<td>RE</td>
<td>Y Y V</td>
<td>X Y Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dqua</td>
<td>DFP Quantize</td>
<td>Z23</td>
<td>FRT, FRA, RMC</td>
<td>Y N</td>
<td>RE</td>
<td>Y Y V</td>
<td>X Y Y</td>
<td>Y</td>
<td></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</td>
<td>RE</td>
<td>Y Y V</td>
<td>X Y Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>drmd</td>
<td>DFP Reround</td>
<td>Z23</td>
<td>FRT, FRA(FIX), FRB, RMC</td>
<td>Y N</td>
<td>RE</td>
<td>Y Y V</td>
<td>X Y Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>drmdq</td>
<td>DFP Reround Quad</td>
<td>Z23</td>
<td>FRTP, FRAP(FIX), FRBP, RMC</td>
<td>Y N</td>
<td>RE</td>
<td>Y Y V</td>
<td>X Y Y</td>
<td>Y</td>
<td></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</td>
<td>RE</td>
<td>Y Y V</td>
<td>X Y Y</td>
<td>Y</td>
<td></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</td>
<td>RE</td>
<td>Y Y V</td>
<td>X Y Y</td>
<td>Y</td>
<td></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</td>
<td>RE</td>
<td>Y Y V</td>
<td>Y# Y Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></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</td>
<td>RE</td>
<td>Y Y V</td>
<td>Y# Y Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></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</td>
<td>RE</td>
<td>Y V</td>
<td>Y# Y Y</td>
<td>Y</td>
<td></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</td>
<td>RE</td>
<td>Y Y V</td>
<td>Y# Y Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dscp</td>
<td>DFP Round To DFP Short</td>
<td>X</td>
<td>FRT (DFP Short), FRB</td>
<td>N Y</td>
<td>RE</td>
<td>Y V</td>
<td>O U X</td>
<td>Y Y Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>drdpq</td>
<td>DFP Round To DFP Long</td>
<td>X</td>
<td>FRT, FRBp</td>
<td>Y N</td>
<td>RE</td>
<td>Y Y V</td>
<td>O U X</td>
<td>Y Y Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dclfpxq</td>
<td>DFP Convert From Fixed Quad</td>
<td>X</td>
<td>FRT, FRBp (DFP Short)</td>
<td>N Y</td>
<td>RE</td>
<td>Y V</td>
<td>O U X</td>
<td>Y Y Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dclfix</td>
<td>DFP Convert To Fixed</td>
<td>X</td>
<td>FRT (FIX), FRB</td>
<td>Y N</td>
<td>-</td>
<td>U U V</td>
<td>X Y</td>
<td>- Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dclfixq</td>
<td>DFP Convert To Fixed Quad</td>
<td>X</td>
<td>FRT (FIX), FRBP</td>
<td>Y N</td>
<td>-</td>
<td>U U V</td>
<td>X Y</td>
<td>- Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ddedpol</td>
<td>DFP Decode DPD To BCD</td>
<td>X</td>
<td>SP, FRT(BCD), FRB</td>
<td>N - -</td>
<td>N</td>
<td>N</td>
<td>- -</td>
<td>Y</td>
<td></td>
<td></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>Vs</th>
<th>G</th>
<th>Encoding</th>
<th>FPRF</th>
<th>FP</th>
<th>PC</th>
<th>CC</th>
<th>Exception</th>
<th>V</th>
<th>Z</th>
<th>O</th>
<th>U</th>
<th>X</th>
<th>FR/F</th>
<th>IE</th>
<th>R</th>
</tr>
</thead>
<tbody>
<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>-</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>denbcd</td>
<td>DFP Encode BCD To DPD</td>
<td>X</td>
<td>S, FRT, FRB (BCD)</td>
<td>-</td>
<td>N</td>
<td>RE</td>
<td>Y</td>
<td>Y</td>
<td>V</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>denbcdq</td>
<td>DFP Encode BCD To DPD 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>V</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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>-</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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>-</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>diex</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>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>diexq</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>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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>-</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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>-</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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>-</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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>-</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Explanation:**

1. FI and FR are set to zeros for these instructions.
2. Not applicable.

A unique definition of the FPSCR_FPRF field is provided for the instruction.

- These are the only instructions that may generate an SNaN and also set the FPSCR_FPRF field. Since the BFP FPSCR_FPRF field does not include a code for SNaN, these instructions cause the need for redefining the FPSCR_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.

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 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.

Undefined

Z A zero-divide exception may be recognized.

Figure 96. Decimal Floating-Point Instructions Summary (Continued)
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.

- \( x.\text{bit}[y] \)
  Return the contents of bit \( y \) of \( x \).

- \( x.\text{bit}[y:z] \)
  Return the contents of bits \( y:z \) of \( x \).

- \( x.\text{nibble}[y] \)
  Return the contents of the 4-bit nibble element \( y \) of \( x \).

- \( x.\text{nibble}[y:z] \)
  Return the contents of the nibble elements \( y:z \) of \( x \).

- \( x.\text{byte}[y] \)
  Return the contents of byte element \( y \) of \( x \).

- \( x.\text{byte}[y:z] \)
  Return the contents of byte elements \( y:z \) of \( x \).

- \( x.\text{halfword}[y] \)
  Return the contents of halfword element \( y \) of \( x \).

- \( x.\text{halfword}[y:z] \)
  Return the contents of halfword elements \( y:z \) 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 element \( y:z \) of \( x \).

- \( x.\text{doubleword}[y] \)
  Return the contents of doubleword element \( y \) of \( x \).

- \( x.\text{doubleword}[y:z] \)
  Return the contents of doubleword elements \( y:z \) of \( x \).

- \( x \ ? y : z \)
  If the value of \( x \) is true, then the value of \( y \), otherwise the value \( z \).

- \( +\text{int} \)
  Integer addition.

- \( +\text{fp} \)
  Floating-point addition.

- \( -\text{fp} \)
  Floating-point subtraction.

- \( \text{sui} \)
  Multiplication of a signed-integer (first operand) by an unsigned-integer (second operand).

- \( \text{fp} \)
  Floating-point multiplication.

- \( =\text{int} \)
  Integer equals relation.

- \( =\text{fp} \)
  Floating-point equals relation.

- \( <\text{ui}, \leq\text{ui}, >\text{ui}, \geq\text{ui} \)
  Unsigned-integer comparison relations.

- \( <\text{si}, \leq\text{si}, >\text{si}, \geq\text{si} \)
  Signed-integer comparison relations.

- \( <\text{fp}, \leq\text{fp}, >\text{fp}, \geq\text{fp} \)
  Floating-point comparison relations.
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.

x +bcd 1
Increments the magnitude of the packed decimal value x by 1.

x << y
Result of shifting x left by y bits, filling vacated bits with zeros.

```
b ← LENGTH(x)
result ← (y < b) ? (x0:31 || y0) : b0
```

x >>ui y
Result of shifting x right by y bits, filling vacated bits with zeros.

```
b ← LENGTH(x)
result ← (y < b) ? (y0 || x0:(b-y)-1) : b0
```

x >> y
Result of shifting x right by y bits, filling vacated bits with copies of bit 0 (sign bit) of x.

```
b ← LENGTH(x)
result ← (y < b) ? (y0 || x0:(b-y)-1) : bx0
```

x <<< y
Result of rotating x left by y bits.

```
b ← LENGTH(x)
result ← x0:31 || x0:y-1
```

x >>> y
Returns the contents of x rotated right by y bits.

Chop(x, y)
Result of extending the right-most y bits of x on the left with zeros.

```
result ← x & (1<y)-1
```

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
    VSCRSAT ← 1
else if (x > z) then
    result ← z
    VSCRSAT ← 1
else result ← x
```

ConvertSitoBCD(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.

```
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)

ConvertBCDtoSI(x)
Let x be a packed decimal value.

Return the value x in signed integer format.

```
result ← 0
scale ← 1
sign ← x.bit[124:127]
x ← x >> 4
```

```
do while (x > 0)
    digit ← x & 0x000F
    result ← result + (digit × scale)
    x ← x >> 4
    scale ← scale × 10
end
```

```
if (sign==0x000B) | (sign==0x000D) then
    result ← ¬result + 1
```

return result
ConvertSPtoSXWsaturate(x, y)
Let \( x \) be a single-precision floating-point value.
Let \( y \) be an unsigned integer value.

\[
\begin{align*}
\text{sign} & \leftarrow x.\text{bit}[0] \\
\text{exp} & \leftarrow x.\text{bit}[1:8] \\
\text{frac.\ bit}[0:22] & \leftarrow x.\text{bit}[9:31] \\
\text{frac.\ bit}[23:30] & \leftarrow 0b0000_0000 \\
& \quad \text{if } (\text{exp}=255) \land (\text{frac} \neq 0) \text{ then return } \{0x0000_0000\} \quad // \text{NaN operand} \\
& \quad \text{if } (\text{exp}=255) \land (\text{frac} = 0) \text{ then do} \quad // \text{infinity operand} \\
& \quad \quad \text{VSCR.SAT} \leftarrow 1 \\
& \quad \quad \text{return } \{(\text{sign} = 1) \, ? \, 0x8000_0000 : \, 0x7FFF_FFFF\} \\
& \quad \end{end{align*}
\]

if \((\text{exp}+Y-127) > 30\) then do \quad // large operand
\quad \text{VSCR.SAT} \leftarrow 1 \\
\quad \text{return } \{(\text{sign} = 1) \, ? \, 0x0000_0000 : \, 0x7FFF_FFFF\}
end

if \((\text{exp}+Y-127) < 0\) then return \{0x0000_0000\} \quad // -1.0 < value < 1.0 \quad // value rounds to 0
\text{significand.\ bit}[0] \leftarrow 0b1 \\
\text{significand.\ bit}[1:31] \leftarrow \text{frac} \\
\quad \text{do } i = 1 \text{ to } 31-(\text{exp}+Y-127) \\
\quad \quad \text{significand} \leftarrow \text{significand} \gg_{\text{ui}} 1 \\
\text{return } \{(\text{sign} = 0) \, ? \, \text{significand} : \, \sim\text{significand} + 1\} \\
\]

ConvertSPtoUXWsaturate(x, y)
Let \( x \) be a single-precision floating-point value.
Let \( y \) be an unsigned integer value.

\[
\begin{align*}
\text{sign} & \leftarrow x.\text{bit}[0] \\
\text{exp} & \leftarrow x.\text{bit}[1:8] \\
\text{frac.\ bit}[0:22] & \leftarrow x.\text{bit}[9:31] \\
\text{frac.\ bit}[23:30] & \leftarrow 0b0000_0000 \\
& \quad \text{if } (\text{exp}=255) \land (\text{frac} \neq 0) \text{ then return } \{0x0000_0000\} \quad // \text{NaN operand} \\
& \quad \text{if } (\text{exp}=255) \land (\text{frac} = 0) \text{ then do} \quad // \text{infinity operand} \\
& \quad \quad \text{VSCR.SAT} \leftarrow 1 \\
& \quad \quad \text{return } \{(\text{sign} = 1) \, ? \, 0x0000_0000 : \, 0xFFFF_FFFF\} \\
& \quad \end{end{align*}
\]

if \((\text{exp}+Y-127) > 31\) then do \quad // large operand
\quad \text{VSCR.SAT} \leftarrow 1 \\
\quad \text{return } \{(\text{sign} = 1) \, ? \, 0x0000_0000 : \, 0xFFFF_FFFF\}
end

if \((\text{exp}+Y-127) < 0\) then return \{0x0000_0000\} \quad // -1.0 < value < 1.0 \quad // value rounds to 0
\text{if( sign=1 ) then do} \quad // negative operand
\text{VSCR.SAT} \leftarrow 1 \\
\text{return } \{0x0000_0000\}
end
\text{significand.\ bit}[0] \leftarrow 0b1 \\
\text{significand.\ bit}[1:31] \leftarrow \text{frac} \\
\quad \text{do } i = 1 \text{ to } 31-(\text{exp}+Y-127) \\
\quad \quad \text{significand} = \text{significand} \gg_{\text{ui}} 1 \\
\text{return } (\text{significand})
ConvertSXWtoSP(x)

Let \( x \) be a 32-bit signed integer value.

\[
\begin{align*}
\text{sign} & \leftarrow x.\text{bit}[0] \\
\text{exp} & \leftarrow 32 + 127 \\
\text{frac}.\text{bit}[0] & \leftarrow x.\text{bit}[0] \\
\text{frac}.\text{bit}[1:32] & \leftarrow x.\text{bit}[0:31] \\
\text{if} \ (\text{frac}==0) & \ \text{return} \ (0x0000_0000) \quad // \text{Zero Operand} \\
\text{if} \ (\text{sign}==1) & \ \text{then} \ \text{frac} = -\text{frac} + 1 \\
\text{do while} \ (\text{frac}.\text{bit}[0]==0) & \\
\quad & \text{frac} \leftarrow \text{frac} \ll 1 \\
\quad & \text{exp} \leftarrow \text{exp} - 1 \\
\text{end} \\
\text{lsb} & \leftarrow \text{frac}.\text{bit}[23] \\
\text{gbit} & \leftarrow \text{frac}.\text{bit}[24] \\
\text{xbit} & \leftarrow \text{frac}.\text{bit}[25:32]!=0 \\
\text{inc} & \leftarrow (\text{lsb} \& \text{gbit}) | (\text{gbit} \& \text{xbit}) \\
\text{frac}.\text{bit}[0:23] & \leftarrow \text{frac}.\text{bit}[0:23] + \text{inc} \\
\text{if} \ (\text{carry}_\text{out}=1) & \ \text{then} \ \text{exp} \leftarrow \text{exp} + 1 \\
\text{result}.\text{bit}[0] & \leftarrow \text{sign} \\
\text{result}.\text{bit}[1:8] & \leftarrow \text{exp} \\
\text{result}.\text{bit}[9:31] & \leftarrow \text{frac}.\text{bit}[1:23] \\
\text{return} \ (\text{result})
\end{align*}
\]

ConvertUXWtoSP(x)

Let \( x \) be a 32-bit unsigned integer value.

\[
\begin{align*}
\text{exp} & \leftarrow 31 + 127 \\
\text{frac} & \leftarrow x.\text{bit}[0:31] \\
\text{if} \ (\text{frac}==0) & \ \text{return} \ (0x0000_0000) \quad // \text{Zero Operand} \\
\text{do while} \ (\text{frac}==0) & \\
\quad & \text{frac} \leftarrow \text{frac} \ll 1 \\
\quad & \text{exp} \leftarrow \text{exp} - 1 \\
\text{end} \\
\text{lsb} & \leftarrow \text{frac}.\text{bit}[23] \\
\text{gbit} & \leftarrow \text{frac}.\text{bit}[24] \\
\text{xbit} & \leftarrow \text{frac}.\text{bit}[25:31]!=0 \\
\text{inc} & \leftarrow (\text{lsb} \& \text{gbit}) | (\text{gbit} \& \text{xbit}) \\
\text{frac}.\text{bit}[0:23] & \leftarrow \text{frac}.\text{bit}[0:23] + \text{inc} \\
\text{if} \ (\text{carry}_\text{out}=1) & \ \text{then} \ \text{exp} \leftarrow \text{exp} + 1 \\
\text{result}.\text{bit}[0] & \leftarrow 0b0 \\
\text{result}.\text{bit}[1:8] & \leftarrow \text{exp} \\
\text{result}.\text{bit}[9:31] & \leftarrow \text{frac}.\text{bit}[1:23] \\
\text{return} \ (\text{result})
\end{align*}
\]

DUP(x,y)

Return the concatenation of \( y \) copies \( x \).

\[
\begin{align*}
\text{DUP}(0b01,4) & = 0b01010101 \\
\text{DUP}(0b001,3) & = 0b001001001
\end{align*}
\]

EXTZ(x)

Result of extending \( x \) on the left with zeros.

\[
\begin{align*}
b & \leftarrow \text{LENGTH}(x) \\
\text{result} & \leftarrow x \& ((1\ll b)\cdot 1)
\end{align*}
\]
InvMixColumns(x)

\[
\text{do } c = 0 \text{ to } 3 \\
\quad \text{result.word}[c].byte[0] = \text{0x0E} \cdot \text{word}[c].byte[0] \oplus \text{0x0B} \cdot \text{word}[c].byte[1] \oplus \text{0x0D} \cdot \text{word}[c].byte[2] \oplus \text{0x09} \cdot \text{word}[c].byte[3] \\
\quad \text{result.word}[c].byte[1] = \text{0x09} \cdot \text{word}[c].byte[0] \oplus \text{0x0E} \cdot \text{word}[c].byte[1] \oplus \text{0x0B} \cdot \text{word}[c].byte[2] \oplus \text{0x0D} \cdot \text{word}[c].byte[3] \\
\quad \text{result.word}[c].byte[2] = \text{0x0D} \cdot \text{word}[c].byte[0] \oplus \text{0x09} \cdot \text{word}[c].byte[1] \oplus \text{0x0E} \cdot \text{word}[c].byte[2] \oplus \text{0x0B} \cdot \text{word}[c].byte[3] \\
\quad \text{result.word}[c].byte[3] = \text{0x0B} \cdot \text{word}[c].byte[0] \oplus \text{0x0D} \cdot \text{word}[c].byte[1] \oplus \text{0x09} \cdot \text{word}[c].byte[2] \oplus \text{0x0E} \cdot \text{word}[c].byte[3] 
\]

end

return(result); 

where “•” is a GF(2^8) multiply, a binary polynomial multiplication reduced by modulo 0x11B.

The GF(2^8) multiply of 0x09•x can be expressed in minimized terms as the following.

\[
\begin{align*}
\text{product.bit}[0] &= \text{x.bit}[0] \oplus \text{x.bit}[3] \\
\text{product.bit}[1] &= \text{x.bit}[1] \oplus \text{x.bit}[4] \oplus \text{x.bit}[0] \\
\text{product.bit}[2] &= \text{x.bit}[2] \oplus \text{x.bit}[5] \oplus \text{x.bit}[0] \oplus \text{x.bit}[1] \\
\text{product.bit}[3] &= \text{x.bit}[3] \oplus \text{x.bit}[6] \oplus \text{x.bit}[1] \oplus \text{x.bit}[2] \\
\text{product.bit}[4] &= \text{x.bit}[4] \oplus \text{x.bit}[7] \oplus \text{x.bit}[2] \\
\text{product.bit}[5] &= \text{x.bit}[5] \oplus \text{x.bit}[0] \oplus \text{x.bit}[1] \\
\text{product.bit}[6] &= \text{x.bit}[6] \oplus \text{x.bit}[0] \oplus \text{x.bit}[1] \oplus \text{x.bit}[2] \\
\text{product.bit}[7] &= \text{x.bit}[7] \oplus \text{x.bit}[0] \oplus \text{x.bit}[2] 
\end{align*}
\]

The GF(2^8) multiply of 0x0B•x can be expressed in minimized terms as the following.

\[
\begin{align*}
\text{product.bit}[0] &= \text{x.bit}[0] \oplus \text{x.bit}[1] \oplus \text{x.bit}[3] \\
\text{product.bit}[1] &= \text{x.bit}[1] \oplus \text{x.bit}[2] \oplus \text{x.bit}[4] \oplus \text{x.bit}[0] \\
\text{product.bit}[2] &= \text{x.bit}[2] \oplus \text{x.bit}[3] \oplus \text{x.bit}[5] \oplus \text{x.bit}[0] \oplus \text{x.bit}[1] \\
\text{product.bit}[3] &= \text{x.bit}[3] \oplus \text{x.bit}[4] \oplus \text{x.bit}[6] \oplus \text{x.bit}[0] \oplus \text{x.bit}[1] \oplus \text{x.bit}[2] \\
\text{product.bit}[4] &= \text{x.bit}[4] \oplus \text{x.bit}[5] \oplus \text{x.bit}[7] \oplus \text{x.bit}[2] \\
\text{product.bit}[5] &= \text{x.bit}[5] \oplus \text{x.bit}[6] \oplus \text{x.bit}[0] \oplus \text{x.bit}[1] \\
\text{product.bit}[6] &= \text{x.bit}[6] \oplus \text{x.bit}[7] \oplus \text{x.bit}[0] \oplus \text{x.bit}[1] \oplus \text{x.bit}[2] \\
\text{product.bit}[7] &= \text{x.bit}[7] \oplus \text{x.bit}[0] \oplus \text{x.bit}[2] 
\end{align*}
\]

The GF(2^8) multiply of 0x0D•x can be expressed in minimized terms as the following.

\[
\begin{align*}
\text{product.bit}[0] &= \text{x.bit}[0] \oplus \text{x.bit}[2] \oplus \text{x.bit}[3] \\
\text{product.bit}[1] &= \text{x.bit}[1] \oplus \text{x.bit}[3] \oplus \text{x.bit}[4] \oplus \text{x.bit}[0] \\
\text{product.bit}[2] &= \text{x.bit}[2] \oplus \text{x.bit}[4] \oplus \text{x.bit}[5] \oplus \text{x.bit}[1] \\
\text{product.bit}[3] &= \text{x.bit}[3] \oplus \text{x.bit}[5] \oplus \text{x.bit}[6] \oplus \text{x.bit}[0] \oplus \text{x.bit}[2] \\
\text{product.bit}[4] &= \text{x.bit}[4] \oplus \text{x.bit}[6] \oplus \text{x.bit}[7] \oplus \text{x.bit}[0] ^ x.bit[2] \\
\text{product.bit}[5] &= \text{x.bit}[5] \oplus \text{x.bit}[7] \oplus \text{x.bit}[1] \\
\text{product.bit}[6] &= \text{x.bit}[6] \oplus \text{x.bit}[7] \oplus \text{x.bit}[0] \oplus \text{x.bit}[1] \oplus \text{x.bit}[2] \\
\text{product.bit}[7] &= \text{x.bit}[7] \oplus \text{x.bit}[0] \oplus \text{x.bit}[2] 
\end{align*}
\]

The GF(2^8) multiply of 0x0E•x can be expressed in minimized terms as the following.

\[
\begin{align*}
\text{product.bit}[0] &= \text{x.bit}[1] \oplus \text{x.bit}[2] \oplus \text{x.bit}[3] \\
\text{product.bit}[1] &= \text{x.bit}[2] \oplus \text{x.bit}[3] \oplus \text{x.bit}[4] \oplus \text{x.bit}[0] \\
\text{product.bit}[2] &= \text{x.bit}[3] \oplus \text{x.bit}[4] \oplus \text{x.bit}[5] \oplus \text{x.bit}[1] \\
\text{product.bit}[3] &= \text{x.bit}[4] \oplus \text{x.bit}[5] \oplus \text{x.bit}[7] \oplus \text{x.bit}[1] \oplus \text{x.bit}[2] \\
\text{product.bit}[4] &= \text{x.bit}[5] \oplus \text{x.bit}[6] \oplus \text{x.bit}[7] \oplus \text{x.bit}[1] \oplus \text{x.bit}[2] \\
\text{product.bit}[5] &= \text{x.bit}[6] \oplus \text{x.bit}[7] \oplus \text{x.bit}[1] \\
\text{product.bit}[6] &= \text{x.bit}[7] \oplus \text{x.bit}[0] \oplus \text{x.bit}[2] \\
\text{product.bit}[7] &= \text{x.bit}[7] \oplus \text{x.bit}[0] \oplus \text{x.bit}[2] 
\end{align*}
\]
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[1].byte[1] = x.word[0].byte[1]
result.word[0].byte[3] = x.word[1].byte[3]
return(result)

InvSubBytes(x)
InvSBOX.byte[256] = { 0x52,0x09,0x6A,0xD5,0x30,0x36,0xA5,0x38,0xBF,0x40,0x03,0xF1,0x0F,0xFD,0x0B,0x5B,0x6F,0x35,0x45,0xF2,0xD3,0xA4,0x62,0x99,0x2A,0x04,0x4C,0xB7,0x4F,0x1C,0xB3,0x28,0x98,0x68,0x51,0xA9,0xF5,0xB0,0x54,0xBB,0xF3,0x1D,0x23,0x25,0x7C,0xF1,0x12,0x32,0x18,0xF6,0/big years

do i = 0 to 15
result.byte[i] = InvSBOX.byte[x.byte[i]]
end
return(result)

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]
end
return(result)

The GF(2^8) multiply of 0x01^x can be expressed in minimized terms as the following.
product.bit[0] = x.bit[1]
product.bit[7] = x.bit[0]
The GF(2^8) multiply of 0x03 \cdot x can be expressed in minimized terms as the following.

\[
\begin{align*}
\text{product.bit}[0] &= \text{x.bit}[0] \oplus \text{x.bit}[1] \\
\text{product.bit}[1] &= \text{x.bit}[1] \oplus \text{x.bit}[2] \\
\text{product.bit}[2] &= \text{x.bit}[2] \oplus \text{x.bit}[3] \\
\text{product.bit}[3] &= \text{x.bit}[3] \oplus \text{x.bit}[4] \oplus \text{x.bit}[0] \\
\text{product.bit}[4] &= \text{x.bit}[4] \oplus \text{x.bit}[5] \oplus \text{x.bit}[0] \\
\text{product.bit}[5] &= \text{x.bit}[5] \oplus \text{x.bit}[6] \\
\text{product.bit}[6] &= \text{x.bit}[6] \oplus \text{x.bit}[7] \oplus \text{x.bit}[0] \\
\text{product.bit}[7] &= \text{x.bit}[7] \oplus \text{x.bit}[0]
\end{align*}
\]

\text{ShiftRows(x)}

\[
\begin{align*}
\text{result.word}[0].\text{byte}[0] &= \text{x.word}[0].\text{byte}[0] \\
\text{result.word}[1].\text{byte}[0] &= \text{x.word}[1].\text{byte}[0] \\
\text{result.word}[2].\text{byte}[0] &= \text{x.word}[2].\text{byte}[0] \\
\text{result.word}[3].\text{byte}[0] &= \text{x.word}[3].\text{byte}[0] \\
\text{result.word}[0].\text{byte}[1] &= \text{x.word}[1].\text{byte}[1] \\
\text{result.word}[1].\text{byte}[1] &= \text{x.word}[2].\text{byte}[1] \\
\text{result.word}[2].\text{byte}[1] &= \text{x.word}[3].\text{byte}[1] \\
\text{result.word}[3].\text{byte}[1] &= \text{x.word}[0].\text{byte}[1] \\
\text{result.word}[0].\text{byte}[2] &= \text{x.word}[2].\text{byte}[2] \\
\text{result.word}[1].\text{byte}[2] &= \text{x.word}[3].\text{byte}[2] \\
\text{result.word}[2].\text{byte}[2] &= \text{x.word}[0].\text{byte}[2] \\
\text{result.word}[3].\text{byte}[2] &= \text{x.word}[1].\text{byte}[2] \\
\text{result.word}[0].\text{byte}[3] &= \text{x.word}[3].\text{byte}[3] \\
\text{result.word}[1].\text{byte}[3] &= \text{x.word}[0].\text{byte}[3] \\
\text{result.word}[2].\text{byte}[3] &= \text{x.word}[1].\text{byte}[3] \\
\text{result.word}[3].\text{byte}[3] &= \text{x.word}[2].\text{byte}[3]
\end{align*}
\]

return(result)

\text{Signed_BCD_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_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.
Signed\_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 \(10^{31}\), ov\_flag is set to 1. Otherwise, ov\_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 eq\_flag, gt\_flag and lt\_flag are set to 0. Otherwise, inv\_flag is set to 0.

SubBytes(x)

\[
\text{SBOX.byte[0:255] = } \{ \\
0x63,0x7C,0x77,0x7B,0x2F,0x6F,0x6F,0xC5,0x30,0x36,0x38,0xFF,0x00,0x02,0x0A,0x0B,0x27,0x28,0x2E,0x2F,0x62,0x6C,0x6C,0x73,0x74,0x75,0x76,0x77,0x78,0x79,0x7A,0x7B,0x7C,0x7D,0x7E,0x7F,0x94,0x95,0x96,0x97,0x98,0x99,0x9A,0x9B,0x9C,0x9D,0x9E,0x9F,0xA0,0xA1,0xA2,0xA3,0xA4,0xA5,0xA6,0xA7,0xA8,0xA9,0xAA,0xAB,0xAC,0xAD,0xAE,0xAF,0xB0,0xB1,0xB2,0xB3,0xB4,0xB5,0xB6,0xB7,0xB8,0xB9,0xBA,0xBB,0xBC,0xBD,0xBE,0xBF,0xC0,0xC1,0xC2,0xC3,0xC4,0xC5,0xC6,0xC7,0xC8,0xC9,0xCA,0xCB,0xCC,0xCD,0xCE,0xCF,0xD0,0xD1,0xD2,0xD3,0xD4,0xD5,0xD6,0xD7,0xD8,0xD9,0xDA,0xDB,0xDC,0xDD,0xDE,0xDF,0xE0,0xE1,0xE2,0xE3,0xE4,0xE5,0xE6,0xE7,0xE8,0xE9,0xEA,0xEB,0xEC,0xED,0xEE,0xEF,0xF0,0xF1,0xF2,0xF3,0xF4,0xF5,0xF6,0xF7,0xF8,0xF9,0xFA,0xFB,0xFC,0xFD,0xFE,0xFF \}
\]

do i = 0 to 15
result.byte[i] = SBOX.byte[x.byte[i]]
end
return(result)

RoundToSPIntCeil(x)
The value \(x\) if \(x\) is a single-precision floating-point integer; otherwise the smallest single-precision floating-point integer that is greater than \(x\).

RoundToSPIntFloor(x)
The value \(x\) if \(x\) is a single-precision floating-point integer; otherwise the largest single-precision floating-point integer that is less than \(x\).

RoundToSPIntNear(x)
The value \(x\) if \(x\) is a single-precision floating-point integer; otherwise the single-precision floating-point integer that is nearest in value to \(x\) (in case of a tie, the even single-precision floating-point integer is used).

RoundToSPIntTrunc(x)
The value \(x\) if \(x\) is a single-precision floating-point integer; otherwise the largest single-precision floating-point integer that is less than \(x\) if \(x>0\), or the smallest single-precision floating-point integer that is greater than \(x\) if \(x<0\).

RoundToNearSP(x)
The single-precision floating-point number that is nearest in value to the infinitely-precise floating-point intermediate result \(x\) (in case of a tie, the single-precision floating-point value with the least-significant bit equal to 0 is used).
ReciprocalEstimateSP(x)
A single-precision floating-point estimate of the reciprocal of the single-precision floating-point number x.

ReciprocalSquareRootEstimateSP(x)
A single-precision floating-point estimate of the reciprocal of the square root of the single-precision floating-point number x.

LogBase2EstimateSP(x)
A single-precision floating-point estimate of the base 2 logarithm of the single-precision floating-point number x.

Power2EstimateSP(x)
A single-precision floating-point estimate of the 2 raised to the power of the single-precision floating-point number x.
6.3 Vector Facility Registers

There are 32 Vector Registers (VRs), each containing 128 bits. See Figure 98. All computations and other data manipulation are performed on data residing in Vector Registers, and results are placed into a VR.

6.3.1 Vector Registers

Depending on the instruction, the contents of a Vector Register 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 Vector Register, as shown in Figure 97. Many instructions perform a given operation in parallel on all elements in a Vector Register. 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 Vector Register.

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 (\texttt{mfvscr} and \texttt{mtvscr}) are provided to move the VSCR from and to a vector register. When moved to or from a vector register, the 32-bit VSCR is right justified in the 128-bit vector register. When moved to a vector register, bits 0:95 of the vector register 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>Vector Non-Java Mode (%)</td>
</tr>
<tr>
<td></td>
<td>This bit controls how denormalized values are handled by Vector Floating-Point instructions.</td>
</tr>
<tr>
<td></td>
<td>Denormalized values are handled as specified by Java and the IEEE standard; see Section 6.6.1.</td>
</tr>
<tr>
<td></td>
<td>If an element in a source VR contains a denormalized value, the value 0 is used instead. If an instruction causes an Underflow Exception, the corresponding element in the target VR is set to 0. In both cases the 0 has the same sign as the denormalized or underflowing value.</td>
</tr>
<tr>
<td>112:126</td>
<td>Reserved</td>
</tr>
<tr>
<td>127</td>
<td>Vector Saturation (\texttt{SAT})</td>
</tr>
</tbody>
</table>

Figure 97. Vector Register elements

Figure 98. Vector Registers

Figure 99. Vector Status and Control Register
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. \texttt{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 \texttt{mtvscr} instruction.

After the \texttt{mtvscr} instruction executes, the result in the target vector register will be architecturally precise. That is, it will reflect all updates to the SAT bit that could have been made by vector instructions logically preceding it in the program flow, and further, it will not reflect any 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 \texttt{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 (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 \texttt{mtvscr} is context synchronizing. This implies that all Vector instructions logically preceding an \texttt{mtvscr} in the program flow will execute in the architectural context (NJ mode) that existed prior to completion of the \texttt{mtvscr}, and that all instructions logically following the \texttt{mtvscr} will execute in the new context (NJ mode) established by the \texttt{mtvscr}.

### 6.3.3 VR Save Register

The VR Save Register (VRSAVE) is a 32-bit register in the fixed-point processor provided for application and operating system use; see Section 3.2.3.
6.4 Vector Storage Access Operations

The Vector Storage Access instructions provide the means by which data can be copied from storage to a Vector Register or from a Vector Register 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 Vector Register or in storage. Instructions are provided that shift and merge the contents of two Vector Registers, such that an unaligned quadword storage operand can be copied between storage and the Vector Registers in a relatively efficient manner.

As shown in Figure 97, the elements in Vector Registers 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 Vector Register, the byte in storage addressed by EA+1 into byte element 1 of the target Vector Register, etc. Similarly, a vector quadword Store instruction for which the EA is quadword-aligned places the contents of byte element 0 of the source Vector Register into the byte in storage addressed by EA, the contents of byte element 1 of the source Vector Register 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 Vector Register or, equivalently, shows the contents that must be in a Vector Register if storing that Vector Register is to produce the storage contents shown in Figure 100.

When an aligned byte, halfword, or word storage operand is loaded into a Vector Register, 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 Vector Register 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 Vector Register 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 Vector Register. An example of this is given in the Programming Note for Vector Splat; see page 257. Another example is to replicate the element across an entire Vector Register 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 101. Vector Register contents for aligned quadword Load or Store

Figure 102. Unaligned quadword storage operand

Figure 103. Vector Register contents
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 Vector Register 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 Vector Register. In Figure 103 Vhi and Vlo are the Vector Registers that will receive the most significant quadword and least significant quadword respectively. VRT is the target Vector Register.

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.

```
# Assumptions:
# Rb != 0 and contents of Rb = 0xB
lvx Vhi,0,Rb   # load MSQ
lvi1 Vp,0,Rb # set permute control vector
addi Rb,Rb,16 # address of LSQ
lvx Vlo,0,Rb   # load LSQ
vperm Vt,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.

```
# Assumptions:
# Rb != 0 and contents of Rb = 0xB
lvx Vhi,0,Rb   # load current MSQ
lver Vp,0,Rb # set permute control vector
addi Rb,Rb,16 # address of LSQ
lvx Vlo,0,Rb   # load current LSQ
vsplit(isb) V1s,-1 # generate the select mask bits
vsplit(isb) V0s,0
vperm Vmask,V0s,V1s,Vp # generate the select mask
vperm Vs,Vs,Vp, Vp # right rotate the data
vsel Vlo, Vs, Vlo, Vmask # insert LSQ component
vsel Vhi, Vs, Vhi, 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 VSCR\textsubscript{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 VSCR\textsubscript{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 VR (mfvscr), bits other than the SAT bit can be cleared in the VR (vand with a constant), the result can be compared to zero setting CR6 (vcmpequb), and a branch can be taken according to whether VSCR\textsubscript{SAT} was set to 1 (Branch Conditional that tests CR field 6).

Testing VSCR\textsubscript{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 VSCR\textsubscript{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 VSCR\textsubscript{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, the SAT bit, which is set to a one (1) anytime any field in a saturating instruction saturates. The SAT bit 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.

The SAT bit 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 vector register (with \texttt{mfvscr}), then masking out bits 0:126 (to clear undefined and reserved bits) and performing a vector compare equal-to unsigned byte w/record (\texttt{vcmpequb}) with zero to get a testable value into the condition register for consumption by a subsequent branch.

Since \texttt{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 VSCRNJ=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 123 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 VSCRSAT 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.

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 VSCRNJ=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 VSCRNJ=1.

- Exceptions that can be caused by a nonzero source value cannot be caused by a denormalized source value when VSCRNJ=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 VRA is a NaN then the result is that NaN else if the element in VRB is a NaN then the result is that NaN else if the element in VRC is a NaN

Programming 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.
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. VSCR_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 inter-
   mediate 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.
   VSCR_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 pos-
   itive number or +infinity, and is 0x8000_0000 if the
   source value is a negative number or -infinity.
   VSCR_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 VSCRNJ=0, the corresponding result is the value
   produced by denormalizing and rounding the inter-
   mediate result.

2. If VSCRNJ=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 1.11.3, “Effective Address Calculation” on page 27. 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 Vector Register. The Load Vector Indexed and Store Vector Indexed instructions transfer an aligned quadword between storage and a Vector Register.

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 register VRT.

--- 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**

\[ \text{lvebx } \text{VRT,RA,RB} \]

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>6</th>
<th>51</th>
<th>46</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>if RA = 0 then b ← 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>else</td>
<td>b ← [RA]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EA</td>
<td>b + [RA]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>eb</td>
<td>EB(63)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VRT ← undefined</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>if Big-Endian byte ordering then</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VRT_{8×eb:8×eb+7} ← MEM(EA,1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>else</td>
<td>VRT_{120-(8×eb):127-(8×eb)} ← MEM(EA,1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Let the effective address (EA) be the sum (RA(0)+(RB).

Let eb be bytes 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 register VRT. The remaining bytes in register VRT 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 register VRT. The remaining bytes in register VRT are set to undefined values.

**Special Registers Altered:**

None

---

**Load Vector Element Halfword Indexed X-form**

\[ \text{lvehx } \text{VRT,RA,RB} \]

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>8</th>
<th>51</th>
<th>16</th>
<th>21</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>if RA = 0 then b ← 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>else</td>
<td>b ← [RA]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EA</td>
<td>b + [RA]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>eb</td>
<td>EB(63)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VRT ← undefined</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>if Big-Endian byte ordering then</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VRT_{8×eb:8×eb+15} ← MEM(EA,2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>else</td>
<td>VRT_{112-(8×eb):127-(8×eb)} ← MEM(EA,2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Let the effective address (EA) be the result of ANDing 0xFFFF_FFFF_FFFF_FFFE with the sum (RA(0)+(RB).

Let eb be bytes 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 register VRT,
- the contents of the byte in storage at address EA+1 are placed into byte eb+1 of register VRT, and
- the remaining bytes in register VRT 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 register VRT,
- the contents of the byte in storage at address EA+1 are placed into byte 14-eb of register VRT, and
- the remaining bytes in register VRT are set to undefined values.

**Special Registers Altered:**

None
Load Vector Element Word Indexed X-form

 lvewx VRT,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>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)) & 0xFFFF_FFFF_FFFF_FFFC

eb ← EA60:63
VRT ← undefined
if Big-Endian byte ordering then
  VRT8×eb:8×eb+31 ← MEM(EA,4)  
else
  VRT96-(8×eb):127-(8×eb) ← MEM(EA,4)

Let the effective address (EA) be the result of ANDing 0xFFFF_FFFF_FFFF_FFFC with the sum (RA|0)+(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 register VRT,
– the contents of the byte in storage at address EA+1 are placed into byte eb+1 of register VRT,
– the contents of the byte in storage at address EA+2 are placed into byte eb+2 of register VRT,
– the contents of the byte in storage at address EA+3 are placed into byte eb+3 of register VRT, and
– the remaining bytes in register VRT are set to undefined values.

If 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 register VRT,
– the contents of the byte in storage at address EA+1 are placed into byte 14-eb of register VRT,
– the contents of the byte in storage at address EA+2 are placed into byte 13-eb of register VRT,
– the contents of the byte in storage at address EA+3 are placed into byte 12-eb of register VRT, and
– the remaining bytes in register VRT are set to undefined values.

Special Registers Altered:
None

Load Vector Indexed X-form

 lvx VRT,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>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)
VRT ← MEM(EA & 0xFFFF_FFFF_FFFF_FFF0, 16)

Let the effective address (EA) be the sum (RA|0)+(RB). The quadword in storage addressed by the result of EA ANDed with 0xFFFF_FFFF_FFFF_FFF0 is loaded into VRT.

Special Registers Altered:
None

Load Vector Indexed Last X-form

 lvxl VRT,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>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)
VRT ← MEM(EA & 0xFFFF_FFFF_FFFF_FFF0, 16)
mark_as_not_likely_to_be_needed_again_anytime_soon(EA)

Let the effective address (EA) be the sum (RA|0)+(RB). The quadword in storage addressed by the result of EA ANDed with 0xFFFF_FFFF_FFFF_FFF0 is loaded into VRT.

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
On some implementations, the hint provided by the \texttt{lvxl} instruction and the corresponding hint provided by the \texttt{stvx}, \texttt{lvpxl}, and \texttt{stvpxl} instructions 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 VRS are stored into the aligned byte, halfword, word, or quadword in storage addressed by EA.

---

**Store Vector Element Byte Indexed X-form**

stvebx VRS,RA,RB

\[
\begin{array}{cccccccc}
31 & VRS & RA & RB & 135 & 8 & 6 & 11 & 16 & 21 & 31 \\
0 & & & & & 0 & 6 & 11 & 16 & 21 & 31 \\
\end{array}
\]

if RA = 0 then b \leftarrow 0
else  b \leftarrow (RA)
EA \leftarrow b + (RB)
eb \leftarrow EA_{60:63}
if Big-Endian byte ordering then
  
  MEM(EA,1) \leftarrow VRS_8 \times eb:8 \times eb+7
else
  
  MEM(EA,1) \leftarrow VRS_{120-(8 \times eb):127-(8 \times eb)}

Let the effective address (EA) be the sum (RA|0)+(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 register VRS 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 register VRS 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 register VRS, software should use Vector Splat to splat the subject byte element before performing the store.

---

**Store Vector Element Halfword Indexed X-form**

stvehx VRS,RA,RB

\[
\begin{array}{cccccccc}
31 & VRS & RA & RB & 167 & 0 & 6 & 11 & 16 & 21 & 31 \\
0 & & & & & 0 & 6 & 11 & 16 & 21 & 31 \\
\end{array}
\]

if RA = 0 then b \leftarrow 0
else  b \leftarrow (RA)
EA \leftarrow b + (RB) & 0xFFFF_FFFF_FFFF_FFFE
eb \leftarrow EA_{60:63}
if Big-Endian byte ordering then
  
  MEM(EA,2) \leftarrow VRS_8 \times eb:8 \times eb+15
else
  
  MEM(EA,2) \leftarrow VRS_{112-(8 \times eb):127-(8 \times eb)}

Let the effective address (EA) be the result of ANDing 0xFFFF_FFFF_FFFF_FFFE with the sum (RA|0)+(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 register VRS are placed in the byte in storage at address EA, and
- the contents of byte eb+1 of register VRS 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 register VRS are placed in the byte in storage at address EA, and
- the contents of byte 14-eb of register VRS 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 register VRS software should use Vector Splat to splat the subject halfword element before performing the store.
Store Vector Element Word Indexed X-form

\texttt{stvewx} \quad \texttt{VRS,RA,RB}

\begin{center}
\begin{tabular}{|c|c|c|c|c|}
\hline
0 & 31 & VRS & RA & RB & 199 \\
\hline
\end{tabular}
\end{center}

\begin{itemize}
\item if RA = 0 then b \leftarrow 0
\item else \quad b \leftarrow [RA]
\item EA \leftarrow [b + (RB)] \& 0xFFFF_FFFF_FFFF_FFFC
\item eb \leftarrow EA_{0:31}
\item if Big-Endian byte ordering then
\item \quad \texttt{MEM(EA,4) \leftarrow VRS_{8\text{x}eb:8\text{x}eb+31}}
\item else
\item \quad \texttt{MEM(EA,4) \leftarrow VRS_{96-(8\text{x}eb):127-(8\text{x}eb)}}
\end{itemize}

Let the effective address (EA) be the result of ANDing 0xFFFF_FFFF_FFFF_FFFC with the sum (RA)(0)+(RB).

Let eb be bits 60:63 of EA.

If Big-Endian byte ordering is used for the storage access,

\begin{itemize}
\item the contents of byte eb of register VRS are placed in the byte in storage at address EA,
\item the contents of byte eb+1 of register VRS are placed in the byte in storage at address EA+1,
\item the contents of byte eb+2 of register VRS are placed in the byte in storage at address EA+2, and
\item the contents of byte eb+3 of register VRS are placed in the byte in storage at address EA+3.
\end{itemize}

If Little-Endian byte ordering is used for the storage access,

\begin{itemize}
\item the contents of byte 15-eb of register VRS are placed in the byte in storage at address EA,
\item the contents of byte 14-eb of register VRS are placed in the byte in storage at address EA+1,
\item the contents of byte 13-eb of register VRS are placed in the byte in storage at address EA+2, and
\item the contents of byte 12-eb of register VRS are placed in the byte in storage at address EA+3.
\end{itemize}

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 register VRS, software should use Vector Splat to splat the subject word element before performing the store.

Store Vector Indexed X-form

\texttt{stvx} \quad \texttt{VRS,RA,RB}

\begin{center}
\begin{tabular}{|c|c|c|c|c|}
\hline
0 & 31 & VRS & RA & RB & 231 \\
\hline
\end{tabular}
\end{center}

\begin{itemize}
\item if RA = 0 then b \leftarrow 0
\item else \quad b \leftarrow [RA]
\item EA \leftarrow [b + (RB)]
\item \texttt{MEM(EA & 0xFFFF_FFFF_FFFF_FFFC, 16) \leftarrow (VRS)}
\end{itemize}

Let the effective address (EA) be the sum (RA)(0)+(RB). The contents of VRS are stored into the quadword in storage addressed by the result of EA ANDed with 0xFFFF_FFFF_FFFF_FFF0.

Special Registers Altered:

None

Store Vector Indexed Last X-form

\texttt{stvxl} \quad \texttt{VRS,RA,RB}

\begin{center}
\begin{tabular}{|c|c|c|c|c|}
\hline
0 & 31 & VRS & RA & RB & 487 \\
\hline
\end{tabular}
\end{center}

\begin{itemize}
\item if RA = 0 then b \leftarrow 0
\item else \quad b \leftarrow [RA]
\item EA \leftarrow [b + (RB)]
\item \texttt{MEM(EA & 0xFFFF_FFFF_FFFF_FFFC, 16) \leftarrow (VRS)}
\item \texttt{mark_as_not Likely_to_be_needed_again_anytime_soon(EA)}
\end{itemize}

Let the effective address (EA) be the sum (RA)(0)+(RB). The contents of VRS are stored into the quadword in storage addressed by the result of EA ANDed with 0xFFFF_FFFF_FFFF_FFF0.

\texttt{stvxl} 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

Programming Note

See the Programming Note for the \texttt{lvxl} instruction on page 243.
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 260). Let X and Y be the contents of register VRA and VRB 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 sh 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 sh bytes.

---

**Load Vector for Shift Left Indexed X-form**

`lvsl VRT,RA,RB`

<table>
<thead>
<tr>
<th>31</th>
<th>VRT</th>
<th>RA</th>
<th>RB</th>
<th>6</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)

sh ← (b + (RB))_{6:13}

switch(sh)

case(0x0): VRT ← 0x000102030405060708090A0B0C0D0E0F

case(0x1): VRT ← 0x0102030405060708090A0B0C0D0E0F10

case(0x2): VRT ← 0x02030405060708090A0B0C0D0E0F11

case(0x3): VRT ← 0x030405060708090A0B0C0D0E0F12

case(0x4): VRT ← 0x0405060708090A0B0C0D0E0F13

case(0x5): VRT ← 0x05060708090A0B0C0D0E0F14

case(0x6): VRT ← 0x060708090A0B0C0D0E0F15

case(0x7): VRT ← 0x0708090A0B0C0D0E0F16

case(0x8): VRT ← 0x08090A0B0C0D0E0F17

case(0x9): VRT ← 0x090A0B0C0D0E0F18

case(0xA): VRT ← 0x0A0B0C0D0E0F19

case(0xB): VRT ← 0x0B0C0D0E0F1A

case(0xC): VRT ← 0x0C0D0E0F1B

case(0xD): VRT ← 0x0D0E0F1C

case(0xE): VRT ← 0x0E0F1D

case(0xF): VRT ← 0x0F1E

Let sh be bits 60:63 of the sum (RA|0)+(RB). Let X be the 32-byte value 0x000 || 0x01 || 0x02 || ... || 0x1E || 0x1F.

Bytes sh to sh+15 of X are placed into VRT.

**Special Registers Altered:** None

---

**Load Vector for Shift Right Indexed X-form**

`lvsr VRT,RA,RB`

<table>
<thead>
<tr>
<th>31</th>
<th>VRT</th>
<th>RA</th>
<th>RB</th>
<th>38</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>16</td>
<td>21</td>
<td>31</td>
</tr>
</tbody>
</table>

if RA = 0 then b ← 0
else b ← (RA)

sh ← (b + (RB))_{6:13}

switch(sh)

case(0x0): VRT ← 0x101112131415161718191A1B1C1D1E1F

case(0x1): VRT ← 0x0F101112131415161718191A1B1C1D1E

case(0x2): VRT ← 0x0E0F101112131415161718191A1B1C1D

case(0x3): VRT ← 0x0D0E0F101112131415161718191A1B1C

case(0x4): VRT ← 0x0C0D0E0F101112131415161718191A1B

case(0x5): VRT ← 0x0B0C0D0E0F101112131415161718191A

case(0x6): VRT ← 0x0A0B0C0D0E0F1011121314151617181A

case(0x7): VRT ← 0x090A0B0C0D0E0F101112131415161718

case(0x8): VRT ← 0x08090A0B0C0D0E0F1011121314151617

case(0x9): VRT ← 0x0708090A0B0C0D0E0F10111213141516

case(0xA): VRT ← 0x060708090A0B0C0D0E0F101112131415

case(0xB): VRT ← 0x05060708090A0B0C0D0E0F1011121314

case(0xC): VRT ← 0x0405060708090A0B0C0D0E0F10111213

case(0xD): VRT ← 0x030405060708090A0B0C0D0E0F10

case(0xE): VRT ← 0x02030405060708090A0B0C0D0E0F

case(0xF): VRT ← 0x0102030405060708090A0B0C0D0E0F

Let sh be bits 60:63 of the sum (RA|0)+(RB). Let X be the 32-byte value 0x000 || 0x01 || 0x02 || ... || 0x1E || 0x1F.

Bytes 16-sh to 31-sh of X are placed into VRT.

**Special Registers Altered:** None

---

**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 Vector Register left (`lvsl`) or right (`lvsr`) by sh bytes. For rotating, the Vector Register to be rotated should be specified as both register VRA and VRB for `vperm`. For shifting left, VRB for `vperm` should be a register containing all zeros and VRA should contain the value to be shifted, and vice versa for shifting right.

**Programming Note**

Examples of uses of `lvsl`, `lvsr`, and `vperm` to load and store unaligned data are given in Section 6.4.1.
6.8 Vector Permute and Formatting Instructions

6.8.1 Vector Pack and Unpack Instructions

**Vector Pack Pixel VX-form**

### VPkpx

<table>
<thead>
<tr>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>6</td>
<td>782</td>
</tr>
</tbody>
</table>

\[
do \ i = 0 \ to \ 63 \ by \ 16 \ \\
\hspace{1cm} VR[VRT]_i \leftarrow VR[VRA]_{i \times 2 + 7} \\
\hspace{1cm} VR[VRT]_{i+1:i+5} \leftarrow VR[VRA]_{i \times 2 + 8:i \times 2 + 12} \\
\hspace{1cm} VR[VRT]_{i+6:i+10} \leftarrow VR[VRA]_{i \times 2 + 16:i \times 2 + 20} \\
\hspace{1cm} VR[VRT]_{i+11:i+15} \leftarrow VR[VRA]_{i \times 2 + 24:i \times 2 + 28} \\
\hspace{1cm} VR[VRT]_{i+64} \leftarrow VR[VRB]_{i \times 2 + 7} \\
\hspace{1cm} VR[VRT]_{i+65:i+69} \leftarrow VR[VRB]_{i \times 2 + 8:i \times 2 + 12} \\
\hspace{1cm} VR[VRT]_{i+70:i+74} \leftarrow VR[VRB]_{i \times 2 + 16:i \times 2 + 20} \\
\hspace{1cm} VR[VRT]_{i+75:i+79} \leftarrow VR[VRB]_{i \times 2 + 24:i \times 2 + 28} \\
\]

Let the source vector be the concatenation of the contents of \( VR[VRA] \) followed by the contents of \( VR[VRB] \).

For each integer value \( i \) from 0 to 7, do the following.

Word element \( i \) in the source vector is packed to produce a 16-bit value as described below.

- Bit 7 of the first byte (bit 7 of the word)
- Bits 0:4 of the second byte (bits 8:12 of the word)
- Bits 0:4 of the third byte (bits 16:20 of the word)
- Bits 0:4 of the fourth byte (bits 24:28 of the word)

The result is placed into halfword element \( i \) of \( VR[VRT] \).

**Special Registers Altered:**

None

**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 Doubleword Signed Saturate VX-form**

### VPksdss

<table>
<thead>
<tr>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>1486</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>6</td>
<td>16</td>
<td>31</td>
</tr>
</tbody>
</table>

\[
\text{src.qword}[0] \leftarrow VR[VRA] \\
\text{src.qword}[1] \leftarrow VR[VRB] \\
\do \ i = 0 \ to \ 3 \ \\
\hspace{1cm} VR[VRT].\text{word}[i] \leftarrow \text{Chop( Clamp( EXTS( src.dword[i]), -2^{31}, 2^{31}-1 ), 32 )} \\
\]

Let doubleword elements 0 and 1 of \( src \) be the contents of \( VR[VRA] \).

Let doubleword elements 2 and 3 of \( src \) be the contents of \( VR[VRB] \).

For each integer value \( i \) from 0 to 3, do the following.

The signed integer value in doubleword element \( i \) of \( src \) is placed into word element \( i \) of \( VR[VRT] \) in signed integer format.

- If the value is greater than \( 2^{31}-1 \) the result saturates to \( 2^{31}-1 \).
- If the value is less than \( -2^{31} \) the result saturates to \( -2^{31} \).

**Special Registers Altered:**

\( SAT \)
Vector Pack Signed Doubleword
Unsigned Saturate VX-form

vpksdus VRT,VRA,VRB

<table>
<thead>
<tr>
<th>4</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>1358</th>
</tr>
</thead>
</table>

src.qword[0] ← VR[VRA]
src.qword[1] ← VR[VRB]
do i = 0 to 3
    VR[VRT].word[i] ← Chop( Clamp( EXTS(src.dword[i]), 0, 2^{32}-1 ), 32 )
end

Let doubleword elements 0 and 1 of src be the contents of VR[VRA].

Let doubleword elements 2 and 3 of src be the contents of VR[VRB].

For each integer value i from 0 to 3, do the following.
The signed integer value in doubleword element i of src is placed into word element i of VR[VRT] in unsigned integer format.
– If the value is greater than 2^{32}-1 the result saturates to 2^{32}-1.
– If the value is less than 0 the result saturates to 0.

Special Registers Altered:
SAT

Vector Pack Signed Halfword Signed Saturate VX-form

vpkshss VRT,VRA,VRB

<table>
<thead>
<tr>
<th>4</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>398</th>
</tr>
</thead>
</table>

doi=0 to 63 by 8
    src1 ← EXTS( VRA[i\times2:i\times2+15] )
    src2 ← EXTS( VRB[i\times2:i\times2+15] )
    VRT_{i\times8+7} ← Clamp( src1, -128, 127 )_{24:31}
    VRT_{i\times8+7+1} ← Clamp( src2, -128, 127 )_{24:31}
end

Let the source vector be the concatenation of the contents of VRA followed by the contents of VRB.

For each integer value i from 0 to 15, do the following.
Signed-integer halfword element i in the source vector is converted to an signed-integer byte.
– If the value of the element is greater than 127 the result saturates to 127
– If the value of the element is less than -128 the result saturates to -128.

The low-order 8 bits of the result is placed into byte element i of VRT.

Special Registers Altered:
SAT
Vector Pack Signed Halfword Unsigned Saturate VX-form


type-1 VRT, VRA, VRB

\[
\begin{array}{cccccc}
0 & 4 & 6 & 11 & 16 & 21 & 270 \\
\end{array}
\]

\[
\begin{array}{cccccc}
0 & 4 & 6 & 11 & 16 & 21 & 270 \\
\end{array}
\]

Let the source vector be the concatenation of the contents of VRA followed by the contents of VRB.

For each integer value i from 0 to 15, do the following.

Signed-integer halfword element i in the source vector is converted to an unsigned-integer byte.

- If the value of the element is greater than 255 the result saturates to 255
- If the value of the element is less than 0 the result saturates to 0.

The low-order 8 bits of the result is placed into byte element i of VRT.

Special Registers Altered:
SAT

Vector Pack Signed Word Signed Saturate VX-form


type-1 VRT, VRA, VRB

\[
\begin{array}{cccccc}
0 & 4 & 6 & 11 & 16 & 21 & 462 \\
\end{array}
\]

\[
\begin{array}{cccccc}
0 & 4 & 6 & 11 & 16 & 21 & 462 \\
\end{array}
\]

Let the source vector be the concatenation of the contents of VRA followed by the contents of VRB.

For each integer value i from 0 to 7, do the following.

Signed-integer word element i in the source vector is converted to a signed-integer halfword.

- If the value of the element is greater than \(2^{15}-1\) the result saturates to \(2^{15}-1\)
- If the value of the element is less than \(-2^{15}\) the result saturates to \(-2^{15}\).

The low-order 16 bits of the result is placed into halfword element i of VRT.

Special Registers Altered:
SAT
Vector Pack Signed Word Unsigned Saturate VX-form

\[
\text{vpkswus VRT, VRA, VRB}
\]

\[
\begin{array}{ccccccc}
0 & 4 & 6 & 8 & 11 & 16 & 21 & 31 \\
\end{array}
\]

\begin{align*}
\text{do } i & = 0 \text{ to } 63 \text{ by } 16 \\
\text{src1} & \leftarrow \text{EXTS}(\text{VRA}[i\times2:i\times2+31]) \\
\text{src2} & \leftarrow \text{EXTS}(\text{VRB}[i\times2:i\times2+31]) \\
\text{VRT}_i:i+15 & \leftarrow \text{Clamp}(\text{src1}, 0, 2^{16}-1:16:31) \\
\text{VRT}_{i+64:i+71} & \leftarrow \text{Clamp}(\text{src2}, 0, 2^{16}-1:16:31)
\end{align*}

Let the source vector be the concatenation of the contents of VRA followed by the contents of VRB.

For each integer value \( i \) from 0 to 7, do the following.

Signed-integer word element \( i \) in the source vector is converted to an unsigned-integer halfword.

- If the value of the element is greater than \( 2^{16}-1 \) the result saturates to \( 2^{16}-1 \)
- If the value of the element is less than 0 the result saturates to 0.

The low-order 16 bits of the result is placed into halfword element \( i \) of VRT.

Special Registers Altered:

SAT

Vector Pack Unsigned Doubleword Ununsigned Saturate VX-form

\[
\text{vpkudus VRT, VRA, VRB}
\]

\[
\begin{array}{ccccccc}
0 & 4 & 6 & 8 & 11 & 16 & 21 & 31 \\
\end{array}
\]

\begin{align*}
\text{if MSR.VEC then Vector_Unavailable()}
\text{src.qword[0]} & \leftarrow \text{VR}[\text{VRA}] \\
\text{src.qword[1]} & \leftarrow \text{VR}[\text{VRB}]
\text{do } i & = 0 \text{ to } 3 \\
\text{VR}[\text{VRT}.\text{word}[i]] & \leftarrow \text{Chop}( \text{Clamp}(\text{EXTZ}(\text{src.dword}[i]), 0, 2^{32}-1), 32 )
\end{align*}

Let doubleword elements 0 and 1 of \( \text{src} \) be the contents of \( \text{VR}[\text{VRA}] \).

Let doubleword elements 2 and 3 of \( \text{src} \) be the contents of \( \text{VR}[\text{VRB}] \).

For each integer value \( i \) from 0 to 3, do the following.

The unsigned integer value in doubleword element \( i \) of \( \text{src} \) is placed into word element \( i \) of \( \text{VR}[\text{VRT}] \) in unsigned integer format.

- If the value of the element is greater than \( 2^{32}-1 \) the result saturates to \( 2^{32}-1 \)

Special Registers Altered:

SAT

Vector Pack Un Signed Halfword Un unsigned Modulo VX-form

\[
\text{vpkuhum VRT, VRA, VRB}
\]

\[
\begin{array}{ccccccc}
0 & 4 & 6 & 8 & 11 & 16 & 21 & 31 \\
\end{array}
\]

\begin{align*}
\text{if MSR.VEC then Vector_Unavailable()}
\text{src.qword[0]} & \leftarrow \text{VR}[\text{VRA}] \\
\text{src.qword[1]} & \leftarrow \text{VR}[\text{VRB}]
\text{do } i & = 0 \text{ to } 63 \text{ by } 8 \\
\text{VRT}_{i:i+7} & \leftarrow \text{Clamp}(\text{EXTZ}(\text{src.dword}[i]), 0, 2^{8}-1:8:126)
\text{VRT}_{i+64:i+71} & \leftarrow \text{Clamp}(\text{src2}, 0, 2^{8}-1:8:126)
\end{align*}

Let the source vector be the concatenation of the contents of VRA followed by the contents of VRB.

For each integer value \( i \) from 0 to 15, do the following.

The contents of bits 32:63 of doubleword element \( i \) of \( \text{src} \) is placed into word element \( i \) of \( \text{VR}[\text{VRT}] \).

Special Registers Altered:

None
Vector Pack Unsigned Halfword Unsigned Saturate VX-form

\[ \text{vpkuhus } \text{VRT, VRA, VRB} \]

\[
\begin{array}{cccccc}
4 & 6 & 7 & 16 & 31 & 31 \\
\text{do } i=0 \text{ to } 63 \text{ by } 8 \\
\text{src1 } & \leftarrow & \text{EXTZ}(\text{VRA})_{i×2:i×2+15} \\
\text{src2 } & \leftarrow & \text{EXTZ}(\text{VRB})_{i×2:i×2+15} \\
\text{VRT}_{i+17} & \leftarrow & \text{Clamp}(\text{src1}, 0, 255)_{24:31} \\
\text{VRT}_{i+64:i+71} & \leftarrow & \text{Clamp}(\text{src2}, 0, 255)_{24:31} \\
\end{array}
\]

Let the source vector be the concatenation of the contents of VRA followed by the contents of VRB.

For each integer value \( i \) from 0 to 15, do the following.

Unsigned-integer halfword element \( i \) in the source vector is converted to an unsigned-integer byte.

- If the value of the element is greater than 255 the result saturates to 255.

The low-order 8 bits of the result is placed into byte element \( i \) of VRT.

Special Registers Altered:

SAT

Vector Pack Unsigned Word Unsigned Modulo VX-form

\[ \text{vpkuwum } \text{VRT, VRA, VRB} \]

\[
\begin{array}{cccccc}
4 & 6 & 7 & 16 & 31 & 31 \\
\text{do } i=0 \text{ to } 63 \text{ by } 16 \\
\text{src1 } & \leftarrow & \text{EXTZ}(\text{VRA})_{i×2:i×2+31} \\
\text{src2 } & \leftarrow & \text{EXTZ}(\text{VRB})_{i×2:i×2+31} \\
\text{VRT}_{i+16} & \leftarrow & \text{Clamp}(\text{src1}, 0, 2^{16}-1)_{16:31} \\
\text{VRT}_{i+64:i+79} & \leftarrow & \text{Clamp}(\text{src2}, 0, 2^{16}-1)_{16:31} \\
\end{array}
\]

Let the source vector be the concatenation of the contents of VRA followed by the contents of VRB.

For each integer value \( i \) from 0 to 7, do the following.

Unsigned-integer word element \( i \) in the source vector is converted to an unsigned-integer halfword.

- If the value of the element is greater than \( 2^{16}-1 \) the result saturates to \( 2^{16}-1 \).

The low-order 16 bits of the result is placed into halfword element \( i \) of VRT.

Special Registers Altered:

None
Vector Unpack High Pixel VX-form

\[
vupkhp x \quad VRT,VRB
\]

<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>

do i=0 to 63 by 16

\[
VRT_{i \times 2:i \times 2+7} \leftrightarrow \text{EXTS}((VRB)_{i})
\]
\[
VRT_{i \times 2+8:i \times 2+15} \leftrightarrow \text{EXTZ}((VRB)_{i+1:i+5})
\]
\[
VRT_{i \times 2+16:i \times 2+23} \leftrightarrow \text{EXTZ}((VRB)_{i+6:i+10})
\]
\[
VRT_{i \times 2+24:i \times 2+31} \leftrightarrow \text{EXTZ}((VRB)_{i+11:i+15})
\]

end

For each vector element \( i \) from 0 to 3, do the following.

- Halfword element \( i \) in VRB is 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 VRT.

Special Registers Altered:
None

---

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 248.

---

Vector Unpack Low Pixel VX-form

\[
vupkl px \quad VRT,VRB
\]

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>///</th>
<th>VRB</th>
<th>974</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

do i=0 to 63 by 16

\[
VRT_{i \times 2:i \times 2+7} \leftrightarrow \text{EXTS}((VRB)_{i+64})
\]
\[
VRT_{i \times 2+8:i \times 2+15} \leftrightarrow \text{EXTZ}((VRB)_{i+65:i+69})
\]
\[
VRT_{i \times 2+16:i \times 2+23} \leftrightarrow \text{EXTZ}((VRB)_{i+70:i+74})
\]
\[
VRT_{i \times 2+24:i \times 2+31} \leftrightarrow \text{EXTZ}((VRB)_{i+75:i+79})
\]

end

For each vector element \( i \) from 0 to 3, do the following.

- Halfword element \( i+4 \) in VRB is 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 VRT.

Special Registers Altered:
None

---

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).
Vector Unpack High Signed Byte VX-form

\[ \text{vupkhsb} \quad \text{VRT,VRB} \]

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>///</th>
<th>VRB</th>
<th>526</th>
</tr>
</thead>
</table>

\[
\begin{align*}
\text{do } i &= 0 \text{ to } 63 \text{ by } 8 \\
\text{VRT}[i+2\times i+15] &= \text{EXTS}((\text{VRB})[i:i+7])
\end{align*}
\]

end

For each vector element \( i \) from 0 to 7, do the following.

Signed-integer byte element \( i \) in VRB is sign-extended to produce a signed-integer halfword and placed into halfword element \( i \) in VRT.

Special Registers Altered:
None

Vector Unpack Low Signed Byte VX-form

\[ \text{vupklsb} \quad \text{VRT,VRB} \]

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>///</th>
<th>VRB</th>
<th>654</th>
</tr>
</thead>
</table>

\[
\begin{align*}
\text{do } i &= 0 \text{ to } 63 \text{ by } 8 \\
\text{VRT}[i+2\times i+15] &= \text{EXTS}((\text{VRB})[i+64:i+71])
\end{align*}
\]

end

For each vector element \( i \) from 0 to 7, do the following.

Signed-integer byte element \( i+8 \) in VRB is sign-extended to produce a signed-integer halfword and placed into halfword element \( i \) in VRT.

Special Registers Altered:
None

Vector Unpack High Signed Halfword VX-form

\[ \text{vupkhs} \quad \text{VRT,VRB} \]

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>///</th>
<th>VRB</th>
<th>590</th>
</tr>
</thead>
</table>

\[
\begin{align*}
\text{do } i &= 0 \text{ to } 63 \text{ by } 16 \\
\text{VRT}[i+2\times i+31] &= \text{EXTS}((\text{VRB})[i:i+15])
\end{align*}
\]

end

For each vector element \( i \) from 0 to 3, do the following.

Signed-integer halfword element \( i \) in VRB is sign-extended to produce a signed-integer word and placed into word element \( i \) in VRT.

Special Registers Altered:
None

Vector Unpack Low Signed Halfword VX-form

\[ \text{vupklsh} \quad \text{VRT,VRB} \]

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>///</th>
<th>VRB</th>
<th>718</th>
</tr>
</thead>
</table>

\[
\begin{align*}
\text{do } i &= 0 \text{ to } 63 \text{ by } 16 \\
\text{VRT}[i+2\times i+31] &= \text{EXTS}((\text{VRB})[i+64:i+79])
\end{align*}
\]

end

For each vector element \( i \) from 0 to 3, do the following.

Signed-integer halfword element \( i+4 \) in VRB is sign-extended to produce a signed-integer word and placed into word element \( i \) in VRT.

Special Registers Altered:
None

Vector Unpack High Signed Word VX-form

\[ \text{vupksw} \quad \text{VRT,VRB} \]

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>///</th>
<th>VRB</th>
<th>1614</th>
</tr>
</thead>
</table>

\[
\begin{align*}
\text{VR[VRT].dword[0]} &= \text{Chop( EXTS(\text{VR[VRB].word[0]}), 64 )} \\
\text{VR[VRT].dword[1]} &= \text{Chop( EXTS(\text{VR[VRB].word[1]}), 64 )}
\end{align*}
\]

For each integer value \( i \) from 0 to 1, do the following.

The signed integer value in word element \( i \) of VR[VRB] is sign-extended and placed into doubleword element \( i \) of VR[VRT].

Special Registers Altered:
None

Vector Unpack Low Signed Word VX-form

\[ \text{vupklsw} \quad \text{VRT,VRB} \]

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>///</th>
<th>VRB</th>
<th>1742</th>
</tr>
</thead>
</table>

\[
\begin{align*}
\text{VR[VRT].dword[0]} &= \text{Chop( EXTS(\text{VR[VRB].word[2]}), 64 )} \\
\text{VR[VRT].dword[1]} &= \text{Chop( EXTS(\text{VR[VRB].word[3]}), 64 )}
\end{align*}
\]

For each integer value \( i \) from 0 to 1, do the following.

The signed integer value in word element \( i+2 \) of VR[VRB] is sign-extended and placed into doubleword element \( i \) of VR[VRT].

Special Registers Altered:
None
6.8.2 Vector Merge Instructions

**Vector Merge High Byte VX-form**

```plaintext
vmrghb VRT, VRA, VRB
```

```

<table>
<thead>
<tr>
<th></th>
<th>4</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>do i=0 to 63 by 8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VRT</td>
<td>x2:i×2+7 &amp;= (VRA)</td>
<td>1:i×7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VRT</td>
<td>x2:i×2+15 &amp;= (VRB)</td>
<td>1:i×7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>end</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For each vector element \( i \) from 0 to 7, do the following.

Byte element \( i \) in VRA is placed into byte element \( 2\times i \) in VRT.

Byte element \( i \) in VRB is placed into byte element \( 2\times i+1 \) in VRT.

Special Registers Altered:
None
```

**Vector Merge Low Byte VX-form**

```plaintext
vmrglb VRT, VRA, VRB
```

```

<table>
<thead>
<tr>
<th></th>
<th>4</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>do i=0 to 63 by 8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VRT</td>
<td>x2:i×2+7 &amp;= (VRA)</td>
<td>i+64:i×7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VRT</td>
<td>x2:i×2+15 &amp;= (VRB)</td>
<td>i+64:i×7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>end</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For each vector element \( i \) from 0 to 7, do the following.

Byte element \( i+8 \) in VRA is placed into byte element \( 2\times i \) in VRT.

Byte element \( i+8 \) in VRB is placed into byte element \( 2\times i+1 \) in VRT.

Special Registers Altered:
None
```

**Vector Merge High Halfword VX-form**

```plaintext
vmrghh VRT, VRA, VRB
```

```

<table>
<thead>
<tr>
<th></th>
<th>4</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>do i=0 to 63 by 16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VRT</td>
<td>x2:i×2+15 &amp;= (VRA)</td>
<td>1:i+15</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VRT</td>
<td>x2:i×2+31 &amp;= (VRB)</td>
<td>1:i+15</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>end</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For each vector element \( i \) from 0 to 3, do the following.

Halfword element \( i \) in VRA is placed into halfword element \( 2\times i \) in VRT.

Halfword element \( i \) in VRB is placed into halfword element \( 2\times i+1 \) in VRT.

Special Registers Altered:
None
```

**Vector Merge Low Halfword VX-form**

```plaintext
vmrglh VRT, VRA, VRB
```

```

<table>
<thead>
<tr>
<th></th>
<th>4</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>do i=0 to 63 by 16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VRT</td>
<td>x2:i×2+15 &amp;= (VRA)</td>
<td>i+64:i+79</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VRT</td>
<td>x2:i×2+31 &amp;= (VRB)</td>
<td>i+64:i+79</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>end</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For each vector element \( i \) from 0 to 3, do the following.

Halfword element \( i+4 \) in VRA is placed into halfword element \( 2\times i \) in VRT.

Halfword element \( i+4 \) in VRB is placed into halfword element \( 2\times i+1 \) in VRT.

Special Registers Altered:
None
**Vector Merge High Word VX-form**

\[ \text{vmrghw VRT,VRA,VRB} \]

```
\[
\begin{array}{c|cccc|}
0 & 4 & 6 & 11 & 16 & 21 & 31 & 140 \\
\end{array}
\]
```

\[
\text{do } i=0 \text{ to } 63 \text{ by } 32 \\
\text{VRT}_i:2:i×2+31 \leftarrow \langle \text{VRA} \rangle_{i:i+31} \\
\text{VRT}_{i+32}:2:i×2+31 \leftarrow \langle \text{VRB} \rangle_{i:i+31} \\
\text{end}
\]

For each vector element \( i \) from 0 to 1, do the following.
Word element \( i \) in VRA is placed into word element \( 2 \times i \) in VRT.

Word element \( i \) in VRB is placed into word element \( 2 \times i+1 \) in VRT.

The word elements in the high-order half of VRA are placed, in the same order, into the even-numbered word elements of VRT. The word elements in the high-order half of VRB are placed, in the same order, into the odd-numbered word elements of VRT.

**Special Registers Altered:**
None

---

**Vector Merge Low Word VX-form**

\[ \text{vmrglw VRT,VRA,VRB} \]

```
\[
\begin{array}{c|cccc|}
5 & 4 & 6 & 11 & 16 & 21 & 31 & 396 \\
\end{array}
\]
```

\[
\text{do } i=0 \text{ to } 63 \text{ by } 32 \\
\text{VRT}_i:2,i+64:2:i×2+31 \leftarrow \langle \text{VRA} \rangle_{i+64:i+95} \\
\text{VRT}_{i+32}:2,i+64:2:i×2+31 \leftarrow \langle \text{VRB} \rangle_{i+64:i+95} \\
\text{end}
\]

For each vector element \( i \) from 0 to 1, do the following.
Word element \( i+2 \) in VRA is placed into word element \( 2 \times i \) in VRT.

Word element \( i+2 \) in VRB is placed into word element \( 2 \times i+1 \) in VRT.

**Special Registers Altered:**
None
**Vector Merge Even Word VX-form**

```plaintext
vmrgew 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>1932</th>
</tr>
</thead>
</table>
```

if MSR.VEC=0 then Vector_Unavailable()

VR[VRT].word[0] ← VR[VRA].word[0]
VR[VRT].word[1] ← VR[VRB].word[0]

The contents of word element 0 of VR[VRA] are placed into word element 0 of VR[VRT].

The contents of word element 0 of VR[VRB] are placed into word element 1 of VR[VRT].

The contents of word element 2 of VR[VRA] are placed into word element 2 of VR[VRT].

The contents of word element 2 of VR[VRB] are placed into word element 3 of VR[VRT].

**vmrgew** is treated as a Vector instruction in terms of resource availability.

**Special Registers Altered**

None

**Vector Merge Odd Word VX-form**

```plaintext
vmrgow 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>1676</th>
</tr>
</thead>
</table>
```

if MSR.VEC=0 then Vector_Unavailable()

VR[VRT].word[0] ← VR[VRA].word[1]

The contents of word element 1 of VR[VRA] are placed into word element 0 of VR[VRT].

The contents of word element 1 of VR[VRB] are placed into word element 1 of VR[VRT].

The contents of word element 3 of VR[VRA] are placed into word element 2 of VR[VRT].

The contents of word element 3 of VR[VRB] are placed into word element 3 of VR[VRT].

**vmrgow** is treated as a Vector instruction in terms of resource availability.

**Special Registers Altered**

None
6.8.3 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 Vector Register by a constant).

Vector Splat Byte VX-form

\texttt{vspltb VRT,VRB,UIM}

\begin{verbatim}
<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>UIM</th>
<th>VRB</th>
<th>524</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>6</td>
<td>11</td>
<td>12</td>
</tr>
</tbody>
</table>
\end{verbatim}

\begin{itemize}
\item \( b \leftarrow \text{UIM} \parallel 0b000 \)
\item \text{do } i = 0 \text{ to } 127 \text{ by } 8
\item \( \text{VRT}_{i:i+7} \leftarrow [\text{VRB}]_{b:b+7} \)
\item end
\end{itemize}

For each integer value \( i \) from 0 to 15, do the following.

The contents of byte element UIM in VRB are placed into byte element \( i \) of VRT.

Special Registers Altered:

None

Vector Splat Word VX-form

\texttt{vspltw VRT,VRB,UIM}

\begin{verbatim}
<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>UIM</th>
<th>VRB</th>
<th>652</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>6</td>
<td>11</td>
<td>14</td>
</tr>
</tbody>
</table>
\end{verbatim}

\begin{itemize}
\item \( b \leftarrow \text{UIM} \parallel 0b00000 \)
\item \text{do } i = 0 \text{ to } 127 \text{ by } 32
\item \( \text{VRT}_{i:i+31} \leftarrow [\text{VRB}]_{b:b+31} \)
\item end
\end{itemize}

For each integer value \( i \) from 0 to 3, do the following.

The contents of word element UIM in VRB are placed into word element \( i \) of VRT.

Special Registers Altered:

None

Vector Splat Halfword VX-form

\texttt{vsplth VRT,VRB,UIM}

\begin{verbatim}
<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>UIM</th>
<th>VRB</th>
<th>588</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>6</td>
<td>11</td>
<td>13</td>
</tr>
</tbody>
</table>
\end{verbatim}

\begin{itemize}
\item \( b \leftarrow \text{UIM} \parallel 0b0000 \)
\item \text{do } i = 0 \text{ to } 127 \text{ by } 16
\item \( \text{VRT}_{i:i+15} \leftarrow [\text{VRB}]_{b:b+15} \)
\item end
\end{itemize}

For each integer value \( i \) from 0 to 7, do the following.

The contents of halfword element UIM in VRB are placed into halfword element \( i \) of VRT.

Special Registers Altered:

None
Vector Splat Immediate Signed Byte
VX-form

vspltisb VRT, SIM

<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></td>
<td></td>
<td></td>
<td>780</td>
</tr>
</tbody>
</table>

do i=0 to 127 by 8
   VRT_{i:i+7} ← EXTS(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 VRT.

Special Registers Altered:
None

Vector Splat Immediate Signed Halfword
VX-form

vspltish VRT, SIM

<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></td>
<td></td>
<td></td>
<td>844</td>
</tr>
</tbody>
</table>

do i=0 to 127 by 16
   VRT_{i:i+15} ← EXTS(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 VRT.

Special Registers Altered:
None

Vector Splat Immediate Signed Word
VX-form

vspltisw VRT, SIM

<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></td>
<td></td>
<td></td>
<td>908</td>
</tr>
</tbody>
</table>

do i=0 to 127 by 32
   VRT_{i:i+31} ← EXTS(SIM, 32)
end

For each vector element 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 VRT.

Special Registers Altered:
None
6.8.4 Vector Permute Instruction

The Vector Permute instruction allows any byte in two source Vector Registers to be copied to any byte in the target Vector Register. The bytes in a third source Vector Register specify from which byte in the first two source Vector Registers the corresponding target byte is to be copied. The contents of the third source Vector Register are sometimes referred to as the “permute control vector”.

**Vector Permute VA-form**

\[ \text{vperm} \ VRT, VRA, VRB, VRC \]

<table>
<thead>
<tr>
<th>0</th>
<th>4</th>
<th>VRT</th>
<th>11</th>
<th>VRA</th>
<th>18</th>
<th>VRB</th>
<th>25</th>
<th>VRC</th>
<th>32</th>
</tr>
</thead>
</table>

if MSR.VEC=0 then VectorUnavailable()

src.qword[0] ← VR[VRA]
src.qword[1] ← VR[VRB]

do i = 0 to 15
index ← VR[VRC].byte[i].bit[3:7]
VR[VRT].byte[i] ← src.byte[index]
end

Let the source vector be the concatenation of the contents of VR[VRA] followed by the contents of VR[VRB].

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 VR[VRC].

The contents of byte element \( \text{index} \) of src are placed into byte element \( i \) of VR[VRT].

**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 247 for examples of uses of vperm.

**Vector Permute Right-indexed VA-form**

\[ \text{vpermr} \ VRT, VRA, VRB, VRC \]

<table>
<thead>
<tr>
<th>0</th>
<th>4</th>
<th>VRT</th>
<th>11</th>
<th>VRA</th>
<th>18</th>
<th>VRB</th>
<th>25</th>
<th>VRC</th>
<th>32</th>
<th>59</th>
</tr>
</thead>
</table>

if MSR.VEC=0 then VectorUnavailable()

src.qword[0] ← VR[VRA]
src.qword[1] ← VR[VRB]

do i = 0 to 15
index ← VR[VRC].byte[i].bit[3:7]
VR[VRT].byte[i] ← src.byte[31-index]
end

Let the source vector be the concatenation of the contents of VR[VRA] followed by the contents of VR[VRB].

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 VR[VRC].

The contents of byte element \( 31-\text{index} \) of src are placed into byte element \( i \) of VR[VRT].

**Special Registers Altered:**
None
6.8.5 Vector Select Instruction

**Vector Select VA-form**

```
vsel VRT, VRA, VRB, VRC
```

<table>
<thead>
<tr>
<th></th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>VRC</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

For each bit in \( VR[C] \) that contains the value 0, the corresponding bit in \( VR[A] \) is placed into the corresponding bit of \( VR[T] \). Otherwise, the corresponding bit in \( VR[B] \) is placed into the corresponding bit of \( VR[T] \).

**Special Registers Altered:**

None
6.8.6 Vector Shift Instructions

The Vector Shift instructions rotate or shift the contents of a Vector Register or a pair of Vector Registers 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 Vector Register 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 ≤ count ≤ 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.

A pair of these instructions, specifying the same shift count register, can be used to shift the contents of a Vector Register 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

Programming Note
```

A pair of these instructions, specifying the same shift count register, can be used to shift the contents of a Vector Register 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.
**Vector Shift Left Double by Octet**  
*Immediate VA-form*

`vsldoi VRT,VRA,VRB,SHB`

|   | 0 | 1 | 2 | 3 | 4 | 5 | 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 |
| 4 | VRT | 6 | VRA | 11 | VRB | 16 | / | SHB | 21 | 22 | 26 | 31 | 44 | 0 | 6 | 11 | 16 | 21 | 22 | 26 | 31 |

if MSR.VEC=0 then Vector_Unavailable();

src.qword[0] ← VR[VRA]
src.qword[1] ← VR[VRB]

VR[VRT] ← src.byte[SHB:SHB+15]

Let the source vector be the concatenation of the contents of VR[VRA] followed by the contents of VR[VRB]. Bytes SHB:SHB+15 of the source vector are placed into VR[VRT].

**Special Registers Altered:**

None
### Vector Shift Left VX-form

**vsll**  
VRT, VRA, VRB

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>452</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()
- \( sh \leftarrow VR[VRB] . bit[125:127] \)
- \( t \leftarrow 1 \)
- \( do \ i = 0 \ to \ 15 \)
- \( t \leftarrow t \& (VR[VRB] . byte[i] . bit[5:7] = sh) \)
- \( end \)
- If \( t=1 \) then
- \( VR[VRT] \leftarrow VR[VRA] \ll sh \)
- Else
- \( VR[VRT] \leftarrow \text{undefined} \)

The contents of \( VR[VRA] \) are shifted left by the number of bits specified in bits 125:127 of \( VR[VRB] \).

- Bits shifted out of bit 0 are lost.
- Zeros are supplied to the vacated bits on the right.

The result is placed into \( VR[VRT] \), except if, for any byte element in register \( VR[VRB] \), the low-order 3 bits are not equal to the shift amount, then \( VR[VRT] \) is undefined.

**Special Registers Altered:**

None

### Vector Shift Right VX-form

**vsrr**  
VRT, VRA, VRB

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>708</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()
- \( sh \leftarrow VR[VRB] . bit[125:127] \)
- \( t \leftarrow 1 \)
- \( do \ i = 0 \ to \ 15 \)
- \( t \leftarrow t \& (VR[VRB] . byte[i] . bit[5:7] = sh) \)
- \( end \)
- If \( t=1 \) then
- \( VR[VRT] \leftarrow VR[VRA] \gg sh \)
- Else
- \( VR[VRT] \leftarrow \text{undefined} \)

The contents of \( VR[VRA] \) are shifted right by the number of bits specified in bits 125:127 of \( VR[VRB] \).

- Bits shifted out of bit 127 are lost.
- Zeros are supplied to the vacated bits on the left.

The result is placed into \( VR[VRT] \), except if, for any byte element in register \( VR[VRB] \), the low-order 3 bits are not equal to the shift amount, then \( VR[VRT] \) is undefined.

**Special Registers Altered:**

None

### Vector Shift Left by Octet VX-form

**vslo**  
VRT, VRA, VRB

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>1036</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()
- \( shb \leftarrow VR[VRB] . bit[121:124] \ll 3 \)
- \( VR[VRT] \leftarrow VR[VRA] \ll shb \)

The contents of \( VR[VRA] \) are shifted left by the number of bytes specified in bits 121:124 of \( VR[VRB] \).

- Bytes shifted out of byte 0 are lost.
- Zeros are supplied to the vacated bytes on the right.

The result is placed into \( VR[VRT] \).

**Special Registers Altered:**

None

### Vector Shift Right by Octet VX-form

**vsro**  
VRT, VRA, VRB

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>1100</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()
- \( shb \leftarrow VR[VRB] . bit[121:124] \ll 3 \)
- \( VR[VRT] \leftarrow VR[VRA] \gg shb \)

The contents of \( VR[VRA] \) are shifted right by the number of bytes specified in bits 121:124 of \( VR[VRB] \).

- Bytes shifted out of byte 15 are lost.
- Zeros are supplied to the vacated bytes on the left.

The result is placed into \( VR[VRT] \).

**Special Registers Altered:**

None
### 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 \( V_w \| V_x \) left by the number of bits specified in \( V_y \) and places the high-order 128 bits of the result into \( V_z \).

```
vslv        Vt1, Vw, Vy      # shift high-order reg left
vspltb      Vy, Vy, 15
vsl         Vt1, Vt1, Vy
vsusubm     Vt3, Vw, Vw     # adjust shift count if VR=A
vsro        Vt2, Vx, Vt3    # shift low-order reg right
vspltb      Vt3, Vt3, 15
vor         Vt2, Vt1, Vt2    # merge to get final result
```

### Vector Shift Left Variable VX-form

**vsll**

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>6</th>
<th>VRA</th>
<th>11</th>
<th>VRB</th>
<th>16</th>
<th>21</th>
<th>1860</th>
</tr>
</thead>
</table>

1. \( V_{w} \)-byte[0:15] ← \( VR[VRA] \)

2. \( V_{w} \)-byte[16] ← 0x00

3. \( V_{z} \)-byte[i] ← \( V_{w} \)-byte[i].bit[5:7].bit[sh:sh+7]

   Let bytes 0:15 of \( src \) be the contents of \( VR[VRA] \).

   Let byte 16 of \( src \) be the value 0x00.

   For each integer value \( i \) from 0 to 15, do the following.
   
   Let \( sh \) be the value in bits 5:7 of byte element \( i \) of \( VR[VRB] \).

   The contents of bits \( sh:sh+7 \) of the halfword in byte elements \( i:i+1 \) of \( src \) are placed into byte element \( i \) of \( VR[VRT] \).

### Special Registers Altered:

None

### Vector Shift Right Variable VX-form

**vsr**

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>6</th>
<th>VRA</th>
<th>11</th>
<th>VRB</th>
<th>16</th>
<th>21</th>
<th>1796</th>
</tr>
</thead>
</table>

1. \( V_{z} \)-byte[0] ← 0x00

2. \( V_{z} \)-byte[1:16] ← \( VR[VRA] \)

3. \( V_{z} \)-byte[i] ← \( V_{z} \)-byte[i].bit[5:7].bit[8-sh:15-sh]

   Let bytes 1:16 of \( src \) be the contents of \( VR[VRA] \).

   Let byte 0 of \( src \) be the value 0x00.

   For each integer value \( i \) from 0 to 15, do the following.
   
   Let \( sh \) be the value in bits 5:7 of byte element \( i \) of \( VR[VRB] \).

   The contents of bits \( 8-sh:15-sh \) of the halfword in byte elements \( i:i+1 \) of \( src \) are placed into byte element \( i \) of \( VR[VRT] \).

### Special Registers Altered:

None
Programmable Note

Assume \( v_{SRC} \) 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.

\[
\begin{align*}
\text{# } v_{SRC} &= \{ \text{0bAAAAAAAB, 0bBBBBBBCC, 0bCCCCCDOD, 0bDDDDDEEEE,} \\
\text{# } & \text{0eFFFFFFFF, 0fGGGGGGG, 0ghhhhhhh, 0iiiiiiii,} \\
\text{# } & \text{0jkkkkkkkl, 0lllllllll, 0mmmmmmmmm, 0nnnnnnnnn,} \\
\text{# } & \text{0nnnnnnnnn, 0nnnnnnnnn, 0nnnnnnnnn, 0nnnnnnnnn} \}
\end{align*}
\]

Assume the following registers are pre-loaded as follows,

\[
\begin{align*}
\text{# } v_{SHCNT1} &= \{ \text{0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x07,} \\
\text{# } & \text{0x07, 0x07, 0x07, 0x07, 0x07, 0x07, 0x07, 0x07}; \\
\text{# } v_{SHCNT2} &= \{ \text{0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x07,} \\
\text{# } & \text{0x07, 0x07, 0x07, 0x07, 0x07, 0x07, 0x07, 0x07}; \\
\text{# } v_{SHCNT3} &= \{ \text{0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x07,} \\
\text{# } & \text{0x07, 0x07, 0x07, 0x07, 0x07, 0x07, 0x07, 0x07}; \\
\text{# } v_{MASK} &= \{ \text{0x7F, 0x7F, 0x7F, 0x7F, 0x7F, 0x7F, 0x7F, 0x7F,} \\
\text{# } & \text{0x7F, 0x7F, 0x7F, 0x7F, 0x7F, 0x7F, 0x7F, 0x7F} \}
\end{align*}
\]

The leftmost seven packed 7-bit values can be unpacked into byte elements 0 to 6 using \texttt{vsrv} with \( v_{SHCNT1} \).

\[
\begin{align*}
\text{vsrv } v_{TMP1}, v_{SRC}, v_{SHCNT1} \# v_{TMP1} &= \{ \text{0b0AAAAAAA, 0bABBBBBBB, 0bBCCCCCCC, 0bCDDDDDDD,} \\
\text{# } & \text{0bDEEEEEEE, 0bEFFFFFFF, 0bFGGGGGGG, 0bGHHHHHHI,} \\
\text{# } & \text{0bIIIIIIIJ, 0bJJJJJJKK, 0bKKKKKLLL, 0bLLLLMMMM,} \\
\text{# } & \text{0bMMNNNNNN, 0bNOOOOOOO, 0bPPPPPPPPQ, 0bQQQQQQRR}; \\
\end{align*}
\]

The next seven packed 7-bit values can then be unpacked into byte elements 7 to 13 using \texttt{vsrv} with \( v_{SHCNT2} \).

\[
\begin{align*}
\text{vsrv } v_{TMP2}, v_{TMP1}, v_{SHCNT2} \# v_{TMP2} &= \{ \text{0b0AAAAAAA, 0bABBBBBBB, 0bBCCCCCCC, 0bCDDDDDDD,} \\
\text{# } & \text{0bDEEEEEEE, 0bEFFFFFFF, 0bFGGGGGGG, 0bGHHHHHHH,} \\
\text{# } & \text{0bHIiiiiii, 0bJJJJJJJJ, 0bKKKKKKK, 0bLLLLLLLL,} \\
\text{# } & \text{0bNNNNNNNN, 0bNNNNNNNN, 0bNNNNNNNN, 0bNNNNNNNN}; \\
\end{align*}
\]

The next two packed 7-bit values can then be unpacked into byte elements 14 to 15 using \texttt{vsrv} with \( v_{SHCNT3} \).

\[
\begin{align*}
\text{vsrv } v_{TMP3}, v_{TMP2}, v_{SHCNT3} \# v_{TMP3} &= \{ \text{0b0AAAAAAA, 0bABBBBBBB, 0bBCCCCCCC, 0bCDDDDDDD,} \\
\text{# } & \text{0bDEEEEEEE, 0bEFFFFFFF, 0bFGGGGGGG, 0bGHHHHHHH,} \\
\text{# } & \text{0bHIiiiiii, 0bJJJJJJJJ, 0bKKKKKKK, 0bLLLLLLLL,} \\
\text{# } & \text{0bNNNNNNNN, 0bNNNNNNNN, 0bNNNNNNNN, 0bNNNNNNNN}; \\
\end{align*}
\]

The most-significant bit in each byte element is masked off to produce a vector of sixteen unsigned byte elements.

\[
\begin{align*}
\text{vand } v_{TMP4}, v_{TMP3}, v_{MASK} \# v_{TMP4} &= \{ \text{0b0AAAAAAA, 0b0BBBBBBB, 0b0CCCCCCC, 0b0DDDDDDD,} \\
\text{# } & \text{0b0EEEEEEE, 0b0FFFFFFF, 0b0GGGGGGG, 0b0HHHHHHH,} \\
\text{# } & \text{0b0IIiiiiii, 0b0JJJJJJJJ, 0b0KKKKKKK, 0b0LLLLLLLL,} \\
\text{# } & \text{0b0NNNNNNN, 0b0NNNNNNN, 0b0NNNNNNN, 0b0NNNNNNN}; \\
\end{align*}
\]

The vector of sixteen unsigned byte elements can be further unpacked to two vectors of eight unsigned halfword elements using a \texttt{vupkhsb} and a \texttt{vupklsh}.

\[
\begin{align*}
\text{vupkhsb } v_{TMP5}, v_{TMP4} \# v_{TMP5} &= \{ \text{0b00000000_0AAAAAAA, 0b00000000_0BBBBBBB, \ldots }; \\
\text{vupklsh } v_{TMP6}, v_{TMP4} \# v_{TMP6} &= \{ \text{0b00000000_01111111, 0b00000000_01111111}; \\ \\
\end{align*}
\]

The resultant two vectors of eight unsigned halfword elements can then be further unpacked to four vectors of four unsigned word elements using two \texttt{vupkhs} and two \texttt{vupkl} instructions.

\[
\begin{align*}
\text{vupkhs } v_{RESULT0}, v_{TMP5} \# v_{RESULT0} &= \{ \text{0b00000000_00000000_00000000_0AAAAAAA, \ldots }; \\
\text{vupkhs } v_{RESULT1}, v_{TMP5} \# v_{RESULT1} &= \{ \text{0b00000000_00000000_00000000_0BBBBBBB, \ldots }; \\
\text{vupkhs } v_{RESULT2}, v_{TMP6} \# v_{RESULT2} &= \{ \text{0b00000000_00000000_00000000_01111111, \ldots }; \\
\text{vupkhs } v_{RESULT3}, v_{TMP6} \# v_{RESULT3} &= \{ \text{0b00000000_00000000_00000000_01111111, \ldots }; \\
\end{align*}
\]
6.8.7 Vector Extract Element Instructions

**Vector Extract Unsigned Byte VX-form**

\[ \text{vextractub VRT,VRB,UIM} \]

The contents of byte element \( UIM \) of \( VR[VRB] \) are placed into bits 56:63 of \( VR[VRT] \). The contents of the remaining byte elements of \( VR[VRT] \) are set to 0.

Special Registers Altered:
None

**Vector Extract Unsigned Halfword VX-form**

\[ \text{vextractuh VRT,VRB,UIM} \]

If the value of \( UIM \) is greater than 14, the results are undefined.

Special Registers Altered:
None

**Vector Extract Unsigned Word VX-form**

\[ \text{vextractuw VRT,VRB,UIM} \]

If the value of \( UIM \) is greater than 12, the results are undefined.

Special Registers Altered:
None

**Vector Extract Doubleword VX-form**

\[ \text{vextractd VRT,VRB,UIM} \]

If the value of \( UIM \) is greater than 8, the results are undefined.

Special Registers Altered:
None
### 6.8.8 Vector Insert Element Instructions

#### Vector Insert Byte VX-form

**vinsertb** VRT,VRB,UIM

<table>
<thead>
<tr>
<th>0</th>
<th>4</th>
<th>VRT</th>
<th>/</th>
<th>UIM</th>
<th>16</th>
<th>21</th>
<th>781</th>
</tr>
</thead>
</table>

if MSR.VEC=0 then Vector_Unavailable()


The contents of byte element 7 of VR[VRB] are placed into byte element UIM of VR[VRT]. The contents of the remaining byte elements of VR[VRT] are not modified.

**Special Registers Altered:**
None

#### Vector Insert Halfword VX-form

**vinserth** VRT,VRB,UIM

<table>
<thead>
<tr>
<th>0</th>
<th>4</th>
<th>VRT</th>
<th>/</th>
<th>UIM</th>
<th>16</th>
<th>21</th>
<th>845</th>
</tr>
</thead>
</table>

if MSR.VE=0 then Vector_Unavailable()


The contents of halfword element 3 of VR[VRB] are placed into byte elements UIM:UIM+1 of VR[VRT]. The contents of the remaining byte elements of VR[VRT] are not modified.

If the value of UIM is greater than 14, the results are undefined.

**Special Registers Altered:**
None

#### Vector Insert Word VX-form

**vinsertw** VRT,VRB,UIM

<table>
<thead>
<tr>
<th>0</th>
<th>4</th>
<th>VRT</th>
<th>/</th>
<th>UIM</th>
<th>16</th>
<th>21</th>
<th>909</th>
</tr>
</thead>
</table>

if MSR.VE=0 then Vector_Unavailable()


The contents of word element 1 of VR[VRB] are placed into byte elements UIM:UIM+3 of VR[VRT]. The contents of the remaining byte elements of VR[VRT] are not modified.

If the value of UIM is greater than 12, the results are undefined.

**Special Registers Altered:**
None

#### Vector Insert Doubleword VX-form

**vinsertd** VRT,VRB,UIM

<table>
<thead>
<tr>
<th>0</th>
<th>4</th>
<th>VRT</th>
<th>/</th>
<th>UIM</th>
<th>16</th>
<th>21</th>
<th>973</th>
</tr>
</thead>
</table>

if MSR.VE=0 then Vector_Unavailable()

VR[VRT].byte[UIM:UIM+7] ← VR[VRB].dword[0]

The contents of doubleword element 0 of VR[VRB] are placed into byte elements UIM:UIM+7 of VR[VRT]. The contents of the remaining byte elements of VR[VRT] are not modified.

If the value of UIM is greater than 8, the results are 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 and Write Carry-Out Unsigned Word VX-form

\[ \text{vaddcuw VRT,VRA,VRB} \]

\[
\begin{array}{cccc}
\text{do } i = 0 \text{ to } 127 \text{ by } 32 \\
\text{aop } & \leftarrow & \text{EXTZ}(\text{VRA}_{i:i+31}) \\
\text{bop } & \leftarrow & \text{EXTZ}(\text{VRB}_{i:i+31}) \\
\text{VRT}_{i:i+31} & \leftarrow & \text{Chop}(\{ \text{aop} + \text{int} \text{bop} \} \gg i 32,1) \\
\end{array}
\]

For each integer value \( i \) from 0 to 3, do the following.
Unsigned-integer word element \( i \) in VRA is added to unsigned-integer word element \( i \) in VRB. The carry out of the 32-bit sum is zero-extended to 32 bits and placed into word element \( i \) of VRT.

Special Registers Altered:
None

Vector Add Signed Byte Saturate VX-form

\[ \text{vaddsbs VRT,VRA,VRB} \]

\[
\begin{array}{cccc}
\text{do } i = 0 \text{ to } 127 \text{ by } 8 \\
\text{aop } & \leftarrow & \text{EXTS}(\text{VRA}_{i:i+7}) \\
\text{bop } & \leftarrow & \text{EXTS}(\text{VRB}_{i:i+7}) \\
\text{VRT}_{i:i+7} & \leftarrow & \text{Clamp}(\text{aop} + \text{int} \text{bop}, -128, 127)_{24:31} \\
\end{array}
\]

For each integer value \( i \) from 0 to 15, do the following.
Signed-integer byte element \( i \) in VRA is added to signed-integer byte element \( i \) in VRB. – If the sum is greater than 127 the result saturates to 127.
– If the sum is less than -128 the result saturates to -128.

The low-order 8 bits of the result are placed into byte element \( i \) of VRT.

Special Registers Altered:
SAT

Vector Add Signed Halfword Saturate VX-form

\[ \text{vaddshs VRT,VRA,VRB} \]

\[
\begin{array}{cccc}
\text{do } i = 0 \text{ to } 127 \text{ by } 16 \\
\text{aop } & \leftarrow & \text{EXTS}(\text{VRA}_{i:i+15}) \\
\text{bop } & \leftarrow & \text{EXTS}(\text{VRB}_{i:i+15}) \\
\text{VRT}_{i:i+15} & \leftarrow & \text{Clamp}(\text{aop} + \text{int} \text{bop}, -2^{15}, 2^{15}-1)_{16:31} \\
\end{array}
\]

For each integer value \( i \) from 0 to 7, do the following.
Signed-integer halfword element \( i \) in VRA is added to signed-integer halfword element \( i \) in VRB.
– If the sum is greater than \( 2^{15}-1 \) the result saturates to \( 2^{15}-1 \)
– If the sum is less than \(-2^{15}\) the result saturates to \(-2^{15}\).

The low-order 16 bits of the result are placed into halfword element \( i \) of VRT.

Special Registers Altered:
SAT
**Vector Add Signed Word Saturate**  
**VX-form**

\[
vaddsws \ VRT, VRA, VRB
\]

<table>
<thead>
<tr>
<th></th>
<th>4</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>896</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Do i=0 to 127 by 32

\[
aop \leftarrow \text{EXTS}(VRA[i:i+31])
bop \leftarrow \text{EXTS}(VRB[i:i+31])
VRT[i:i+31] \leftarrow \text{Clamp}(aop + \text{int} bop, -2^{31}, 2^{31}-1)
\]

For each integer value \(i\) from 0 to 3, do the following.
Signed-integer word element \(i\) in \(VRA\) is added to signed-integer word element \(i\) in \(VRB\).

- If the sum is greater than \(2^{31} - 1\) the result saturates to \(2^{31} - 1\).
- If the sum is less than \(-2^{31}\) the result saturates to \(-2^{31}\).

The low-order 32 bits of the result are placed into word element \(i\) of \(VRT\).

**Special Registers Altered:**
SAT

**Vector Add Unsigned Byte Modulo**  
**VX-form**

\[
vaddubm \ VRT, VRA, VRB
\]

<table>
<thead>
<tr>
<th></th>
<th>4</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Do i=0 to 127 by 8

\[
aop \leftarrow \text{EXTZ}(VRA[i:i+7])
bop \leftarrow \text{EXTZ}(VRB[i:i+7])
VRT[i:i+7] \leftarrow \text{Chop}(aop + \text{int} bop, 8)
\]

For each integer value \(i\) from 0 to 15, do the following.
Unsigned-integer byte element \(i\) in \(VRA\) is added to unsigned-integer byte element \(i\) in \(VRB\).

The low-order 8 bits of the result are placed into byte element \(i\) of \(VRT\).

**Special Registers Altered:**
None

**Programming Note**

\textit{vaddubm} can be used for unsigned or signed-integers.

**Vector Add Unsigned Doubleword Modulo**  
**VX-form**

\[
vaddudm \ VRT, VRA, VRB
\]

<table>
<thead>
<tr>
<th></th>
<th>4</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>192</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Do i = 0 to 1

\[
aop \leftarrow VR[VRA].dword[i]
bop \leftarrow VR[VRB].dword[i]
VR[VRT].dword[i] \leftarrow \text{Chop}(aop + \text{int} bop, 64)
\]

For each integer value \(i\) from 0 to 1, do the following.
The integer value in doubleword element \(i\) of \(VR[VRB]\) is added to the integer value in doubleword element \(i\) of \(VR[VRA]\).

The low-order 64 bits of the result are placed into doubleword element \(i\) of \(VR[VRT]\).

**Special Registers Altered:**
None

**Programming Note**

\textit{vaddudm} can be used for signed or unsigned integers.
Vector Add Unsigned Halfword Modulo VX-form

\[ \text{vadduhm } \text{VRT}, \text{VRA}, \text{VRB} \]

<table>
<thead>
<tr>
<th>4</th>
<th>6</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>64</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>

do i=0 to 127 by 16
  aop \leftarrow \text{EXT2(}\text{VRA}_{i:i+15}\text{)}
  bop \leftarrow \text{EXT2(}\text{VRB}_{i:i+15}\text{)}
  \text{VRT}_{i:i+15} \leftarrow \text{Chop( } \text{aop } + \text{int } \text{bop}, 16 \text{ )}
end

For each integer value \( i \) from 0 to 7, do the following.
  Unsigned-integer halfword element \( i \) in VRA is added to unsigned-integer halfword element \( i \) in VRB.

The low-order 16 bits of the result are placed into halfword element \( i \) of VRT.

Special Registers Altered:
  None

Programming Note
  \text{vadduhm} \text{ can be used for unsigned or signed-integers.}

Vector Add Unsigned Word Modulo VX-form

\[ \text{vadduwm } \text{VRT}, \text{VRA}, \text{VRB} \]

<table>
<thead>
<tr>
<th>4</th>
<th>6</th>
<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>
<td>21</td>
<td>31</td>
</tr>
</tbody>
</table>

do i=0 to 127 by 32
  aop \leftarrow \text{EXT2(}\text{VRA}_{i:i+31}\text{)}
  bop \leftarrow \text{EXT2(}\text{VRB}_{i:i+31}\text{)}
  \text{temp} \leftarrow \text{aop } + \text{int } \text{bop}
  \text{VRT}_{i:i+31} \leftarrow \text{Chop( } \text{aop } + \text{int } \text{bop}, 32 \text{ )}
end

For each integer value \( i \) from 0 to 3, do the following.
  Unsigned-integer word element \( i \) in VRA is added to unsigned-integer word element \( i \) in VRB.

The low-order 32 bits of the result are placed into word element \( i \) of VRT.

Special Registers Altered:
  None

Programming Note
  \text{vadduwm} \text{ can be used for unsigned or signed-integers.}
### Vector Add Unsigned Byte Saturate VX-form

**vaddubs**

\[ \text{VRT, VRA, VRB} \]

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>31</td>
<td>31</td>
</tr>
</tbody>
</table>

\[
\begin{array}{l}
do \text{i}=0 \text{ to } 127 \text{ by } 8 \\
aop \leftarrow \text{EXTZ}((\text{VRA})_{i:i+7}) \\
bop \leftarrow \text{EXTZ}((\text{VRB})_{i:i+7}) \\
\text{VRT}_{i:i+7} \leftarrow \text{Clamp}(\text{aop} + \text{int} \ bop, 0, 255)_{24:31} \\
\end{array}
\]

For each integer value \( i \) from 0 to 15, do the following.

Unsigned-integer byte element \( i \) in VRA is added to unsigned-integer byte element \( i \) in VRB.

- If the sum is greater than 255 the result saturates to 255.

The low-order 8 bits of the result are placed into byte element \( i \) of VRT.

**Special Registers Altered:**

SAT

### Vector Add Unsigned Halfword Saturate VX-form

**vadduhw**

\[ \text{VRT, VRA, VRB} \]

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>6</td>
<td>56</td>
<td>31</td>
</tr>
</tbody>
</table>

\[
\begin{array}{l}
do \text{i}=0 \text{ to } 127 \text{ by } 16 \\
aop \leftarrow \text{EXTZ}((\text{VRA})_{i:i+15}) \\
bop \leftarrow \text{EXTZ}((\text{VRB})_{i:i+15}) \\
\text{VRT}_{i:i+15} \leftarrow \text{Clamp}(\text{aop} + \text{int} \ bop, 0, 2^{32}-1)_{16:31} \\
\end{array}
\]

For each integer value \( i \) from 0 to 7, do the following.

Unsigned-integer halfword element \( i \) in VRA is added to unsigned-integer halfword element \( i \) in VRB.

- If the sum is greater than \( 2^{32}-1 \) the result saturates to \( 2^{32}-1 \).

The low-order 32 bits of the result are placed into halfword element \( i \) of VRT.

**Special Registers Altered:**

SAT

### Vector Add Unsigned Word Saturate VX-form

**vadduw**

\[ \text{VRT, VRA, VRB} \]

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>6</td>
<td>576</td>
<td>31</td>
</tr>
</tbody>
</table>

\[
\begin{array}{l}
do \text{i}=0 \text{ to } 127 \text{ by } 32 \\
aop \leftarrow \text{EXTZ}((\text{VRA})_{i:i+31}) \\
bop \leftarrow \text{EXTZ}((\text{VRB})_{i:i+31}) \\
\text{VRT}_{i:i+31} \leftarrow \text{Clamp}(\text{aop} + \text{int} \ bop, 0, 2^{64}-1) \\
\end{array}
\]

For each integer value \( i \) from 0 to 3, do the following.

Unsigned-integer word element \( i \) in VRA is added to unsigned-integer word element \( i \) in VRB.

- If the sum is greater than \( 2^{64}-1 \) the result saturates to \( 2^{64}-1 \).

The low-order 64 bits of the result are placed into word element \( i \) of VRT.

**Special Registers Altered:**

SAT
Vector Add Unsigned Quadword Modulo

**VX-form**

\[ \text{vadduqm } \text{VRT, VRA, VRB} \]

<table>
<thead>
<tr>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>256</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>6</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

If MSR.VEC=0 then Vector_Unavailable()

\[
\begin{align*}
\text{src1} & \leftarrow \text{VR}[\text{VRA}] \\
\text{src2} & \leftarrow \text{VR}[\text{VRB}] \\
\text{sum} & \leftarrow \text{EXTZ}(\text{src1}) + \text{EXTZ}(\text{src2}) \\
\text{VRT} & \leftarrow \text{Chop}(\text{sum}, 128)
\end{align*}
\]

Let src1 be the integer value in VR[VRT].

Let src2 be the integer value in VR[VRT].

src1 and src2 can be signed or unsigned integers.

The rightmost 128 bits of the sum of src1 and src2 are placed into VR[VRT].

Special Registers Altered:
None

Vector Add Extended Unsigned Quadword Modulo

**VA-form**

\[ \text{vaddeuqm } \text{VRT, VRA, VRB, VRC} \]

<table>
<thead>
<tr>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>VRC</th>
<th>60</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>6</td>
<td>16</td>
<td>21</td>
<td>26</td>
</tr>
</tbody>
</table>

If MSR.VEC=0 then Vector_Unavailable()

\[
\begin{align*}
\text{src1} & \leftarrow \text{VR}[\text{VRA}] \\
\text{src2} & \leftarrow \text{VR}[\text{VRB}] \\
\text{cin} & \leftarrow \text{VR}[\text{VRC}].\text{bit}[127] \\
\text{sum} & \leftarrow \text{EXTZ}(\text{src1}) + \text{EXTZ}(\text{src2}) + \text{EXTZ}(\text{cin}) \\
\text{VRT} & \leftarrow \text{Chop}(\text{sum}, 128)
\end{align*}
\]

Let src1 be the integer value in VR[VRT].

Let src2 be the integer value in VR[VRT].

Let cin be the integer value in bit 127 of VR[VRT].

src1 and src2 can be signed or unsigned integers.

The rightmost 128 bits of the sum of src1, src2, and cin are placed into VR[VRT].

Special Registers Altered:
None

Vector Add & write Carry Unsigned Quadword

**VX-form**

\[ \text{vaddcuq } \text{VRT, VRA, VRB} \]

<table>
<thead>
<tr>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>256</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>6</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

If MSR.VEC=0 then Vector_Unavailable()

\[
\begin{align*}
\text{src1} & \leftarrow \text{VR}[\text{VRA}] \\
\text{src2} & \leftarrow \text{VR}[\text{VRB}] \\
\text{sum} & \leftarrow \text{EXTZ}(\text{src1}) + \text{EXTZ}(\text{src2}) \\
\text{VRT} & \leftarrow \text{Chop}(\text{EXTZ}(\text{Chop}(\text{sum} \gg 128, 1)), 128)
\end{align*}
\]

Let src1 be the integer value in VR[VRT].

Let src2 be the integer value in VR[VRT].

The carry out of the sum of src1 and src2 is placed into VR[VRT].

Special Registers Altered:
None

Vector Add Extended & write Carry Unsigned Quadword

**VA-form**

\[ \text{vaddecuq } \text{VRT, VRA, VRB, VRC} \]

<table>
<thead>
<tr>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>VRC</th>
<th>61</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>6</td>
<td>16</td>
<td>21</td>
<td>26</td>
</tr>
</tbody>
</table>

If MSR.VEC=0 then Vector_Unavailable()

\[
\begin{align*}
\text{src1} & \leftarrow \text{VR}[\text{VRA}] \\
\text{src2} & \leftarrow \text{VR}[\text{VRB}] \\
\text{cin} & \leftarrow \text{VR}[\text{VRC}].\text{bit}[127] \\
\text{sum} & \leftarrow \text{EXTZ}(\text{src1}) + \text{EXTZ}(\text{src2}) + \text{EXTZ}(\text{cin}) \\
\text{VRT} & \leftarrow \text{Chop}(\text{EXTZ}(\text{Chop}(\text{sum} \gg 128, 1)), 128)
\end{align*}
\]

Let src1 be the integer value in VR[VRT].

Let src2 be the integer value in VR[VRT].

Let cin be the integer value in bit 127 of VR[VRT].

src1 and src2 can be signed or unsigned integers.

The carry out of the sum of src1, src2, and cin are placed into VR[VRT].

Special Registers Altered:
None
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
vadcduq vC3, vA3, vB3  # carry out of bit 384 of sum
vadddeuqm vS2, vA2, vB2, vC3  # bits 256:383 of sum
vadddecuq vC2, vA2, vB2, vC3  # carry out of bit 256 of sum
vadddeuqm vS1, vA1, vB1, vC2  # bits 128:255 of sum
vadddecuq vC1, vA1, vB1, vC2  # carry out of bit 128 of sum
vadddeuqm vS0, vA0, vB0, vC1  # bits 0:127 of sum
```
6.9.1.2 Vector Integer Subtract Instructions

**Vector Subtract and Write Carry-Out**

*Unsigned Word VX-form*

**vsubcuw**  
VRT, VRA, VRB

<table>
<thead>
<tr>
<th>i</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>1408</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>8</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

```
do i = 0 to 127 by 32
    aop ← EXTZ((VRA)i:i+31)
    bop ← EXTZ((VRB)i:i+31)
    temp ← (aop _int ^ bop _int 1) >> 32
    VRTi:i+31 ← temp & 0x0000_0001
end```

For each integer value i from 0 to 3, do the following. Unsigned-integer word element i in VRB is subtracted from unsigned-integer word element i in VRA. 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 VRT.

**Special Registers Altered:**
None

**Vector Subtract Signed Byte Saturate**

*VX-form*

**vsubsb**  
VRT, VRA, VRB

<table>
<thead>
<tr>
<th>i</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>1792</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

```
do i = 0 to 127 by 8
    aop ← EXTS((VRA)i:i+7)
    bop ← EXTS((VRB)i:i+7)
    temp ← Clamp(aop _int ^ bop _int 1, -128, 127)_24:31
    VRTi:i+7 ← temp
end```

For each integer value i from 0 to 15, do the following. Signed-integer byte element i in VRB is subtracted from signed-integer byte element i in VRA.

- If the intermediate result is greater than 127 the result saturates to 127.
- If the intermediate result is less than -128 the result saturates to -128.

The low-order 8 bits of the result are placed into byte element i of VRT.

**Special Registers Altered:**
SAT

**Vector Subtract Signed Halfword Saturate**

*VX-form*

**vsubsh**  
VRT, VRA, VRB

<table>
<thead>
<tr>
<th>i</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>1856</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

```
do i = 0 to 127 by 16
    aop ← EXTS((VRA)i:i+15)
    bop ← EXTS((VRB)i:i+15)
    temp ← Clamp(aop _int ^ bop _int 1, -215, 215-1)_16:31
    VRTi:i+15 ← temp
end```

For each integer value i from 0 to 7, do the following. Signed-integer halfword element i in VRB is subtracted from signed-integer halfword element i in VRA.

- If the intermediate result is greater than 2^{15-1} the result saturates to 2^{15-1}.
- If the intermediate result is less than -2^{15} the result saturates to -2^{15}.

The low-order 16 bits of the result are placed into halfword element i of VRT.

**Special Registers Altered:**
SAT
Vector Subtract Signed Word Saturate
VX-form

vsubsws VRT,VRA,VRB

For each integer value i from 0 to 3, do the following.
Signed-integer word element i in VRB is
subtracted from signed-integer word element i in
VRA.

– 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 low-order 32 bits of the result are placed into
word element i of VRT.

Special Registers Altered:
SAT
Vector Subtract Unsigned Byte Modulo VX-form

\texttt{vsububm} VRT, VRA, VRB

\begin{tabular}{|c|c|c|c|c|c|}
\hline
4 & 6 & 11 & 16 & 21 & 1024 \\
\hline
\end{tabular}

\begin{verbatim}
do i=0 to 127 by 8
  aop ← \text{EXTZ}(VRA)_{\text{i:i+7}}
  bop ← \text{EXTZ}(VRB)_{\text{i:i+7}}
  VRT_{\text{i:i+7}} ← \text{Chop}( \text{aop} +\text{int } \text{¬bop } +\text{int 1}, 8 )
\end{verbatim}

For each integer value \(i\) from 0 to 15, do the following.
Signed-integer byte element \(i\) in VRB is
subtracted from signed-integer byte element \(i\) in
VRA. The low-order 8 bits of the result are placed
into byte element \(i\) of VRT.

Special Registers Altered:
None

Programming Note
\texttt{vsububm} can be used for signed or unsigned inte-
gers.

Vector Subtract Unsigned Doubleword Modulo VX-form

\texttt{vsubudm} VRT, VRA, VRB

\begin{tabular}{|c|c|c|c|c|c|}
\hline
4 & 6 & 11 & 16 & 21 & 1216 \\
\hline
\end{tabular}

\begin{verbatim}
do i=0 to 1
  aop ← VR[VRA].\text{dword}[i]
  bop ← VR[VRB].\text{dword}[i]
  VRT.\text{dword}[i] ← \text{Chop}( \text{aop } +\text{int } \text{¬bop } +\text{int 1}, 64 )
\end{verbatim}

For each integer value \(i\) from 0 to 1, do the following.
The integer value in doubleword element \(i\) of
VRB is subtracted from the integer value in
doubleword element \(i\) of VRA.

The low-order 64 bits of the result are placed into
doubleword element \(i\) of VRT.

Special Registers Altered:
None

Vector Subtract Unsigned Halfword Modulo VX-form

\texttt{vsubuhm} VRT, VRA, VRB

\begin{tabular}{|c|c|c|c|c|c|}
\hline
4 & 6 & 11 & 16 & 21 & 1088 \\
\hline
\end{tabular}

\begin{verbatim}
do i=0 to 127 by 16
  aop ← \text{EXTZ}(VRA)_{\text{i:i+15}}
  bop ← \text{EXTZ}(VRB)_{\text{i:i+15}}
  VRT_{\text{i:i+16}} ← \text{Chop}( \text{aop } +\text{int } \text{¬bop } +\text{int 1}, 16 )
\end{verbatim}

For each integer value \(i\) from 0 to 7, do the following.
Unsigned-integer halfword element \(i\) in VRB is
subtracted from unsigned-integer halfword
element \(i\) in VRA. The low-order 16 bits of the
result are placed into halfword element \(i\) of VRT.

Special Registers Altered:
None

Vector Subtract Unsigned Word Modulo VX-form

\texttt{vsubuwm} VRT, VRA, VRB

\begin{tabular}{|c|c|c|c|c|c|}
\hline
4 & 6 & 11 & 16 & 21 & 1152 \\
\hline
\end{tabular}

\begin{verbatim}
do i=0 to 127 by 32
  aop ← \text{EXTZ}(VRA)_{\text{i:i+31}}
  bop ← \text{EXTZ}(VRB)_{\text{i:i+31}}
  VRT_{\text{i:i+31}} ← \text{Chop}( \text{aop } +\text{int } \text{¬bop } +\text{int 1}, 32 )
\end{verbatim}

For each integer value \(i\) from 0 to 3, do the following.
Unsigned-integer word element \(i\) in VRB is
subtracted from unsigned-integer word element \(i\)
in VRA. The low-order 32 bits of the result are
placed into word element \(i\) of VRT.

Special Registers Altered:
None
Vector Subtract Unsigned Byte Saturate VX-form
vsububs VRT, VRA, VRB

<table>
<thead>
<tr>
<th>i</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>1536</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>21</td>
<td>16</td>
<td>31</td>
</tr>
</tbody>
</table>

\[\text{do } i = 0 \text{ to } 127 \text{ by } 8\]
\[\text{aop } \leftarrow \text{EXTZ}(\text{VRA})_{i:i+7}\]
\[\text{bop } \leftarrow \text{EXTZ}(\text{VRB})_{i:i+7}\]
\[\text{VRT}_{i:i+7} \leftarrow \text{Clamp}(\text{aop} \oplus \text{bop} + 1,0,255)_{24:31}\]
\[\text{end}\]

For each integer value \(i\) from 0 to 15, do the following.
Unsigned-integer byte element \(i\) in VRB is subtracted from unsigned-integer byte element \(i\) in VRA. If the intermediate result is less than 0 the result saturates to 0. The low-order 8 bits of the result are placed into byte element \(i\) of VRT.

Special Registers Altered:
SAT

Vector Subtract Unsigned Halfword Saturate VX-form
vsubuhs VRT, VRA, VRB

<table>
<thead>
<tr>
<th>i</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>1600</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>21</td>
<td>16</td>
<td>31</td>
</tr>
</tbody>
</table>

\[\text{do } i = 0 \text{ to } 127 \text{ by } 16\]
\[\text{aop } \leftarrow \text{EXTZ}(\text{VRA})_{i:i+15}\]
\[\text{bop } \leftarrow \text{EXTZ}(\text{VRB})_{i:i+15}\]
\[\text{VRT}_{i:i+15} \leftarrow \text{Clamp}(\text{aop} \oplus \text{bop} + 1,0,2^{16}-1)_{16:31}\]
\[\text{end}\]

For each integer value \(i\) from 0 to 7, do the following.
Unsigned-integer halfword element \(i\) in VRB is subtracted from unsigned-integer halfword element \(i\) in VRA. If the intermediate result is less than 0 the result saturates to 0. The low-order 16 bits of the result are placed into halfword element \(i\) of VRT.

Special Registers Altered:
SAT

Vector Subtract Unsigned Word Saturate VX-form
vsubuws VRT, VRA, VRB

<table>
<thead>
<tr>
<th>i</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>1664</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>21</td>
<td>16</td>
<td>31</td>
</tr>
</tbody>
</table>

\[\text{do } i = 0 \text{ to } 127 \text{ by } 32\]
\[\text{aop } \leftarrow \text{EXTZ}(\text{VRA})_{i:i+31}\]
\[\text{bop } \leftarrow \text{EXTZ}(\text{VRB})_{i:i+31}\]
\[\text{VRT}_{i:i+31} \leftarrow \text{Clamp}(\text{aop} \oplus \text{bop} + 1,0,2^{32}-1)_{32:31}\]
\[\text{end}\]

For each integer value \(i\) from 0 to 7, do the following.
Unsigned-integer word element \(i\) in VRB is subtracted from unsigned-integer word element \(i\) in VRA. If the intermediate result is less than 0 the result saturates to 0. The low-order 32 bits of the result are placed into word element \(i\) of VRT.

Special Registers Altered:
SAT
### Vector Subtract Unsigned Quadword Modulo VX-form

<table>
<thead>
<tr>
<th>vsubuqm</th>
<th>VRT, VRA, VRB</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>6</td>
</tr>
</tbody>
</table>

```plaintext
if MSR.VEC=0 then Vector_Unavailable();
src1 ← VR[VRA];
src2 ← VR[VRB];
sum ← EXTZ(src1) + EXTZ(~src2) + EXTZ(1);
VR[VRT] ← Chop(sum, 128);
```

Let \( src_1 \) be the integer value in \( VR[VRA] \).
Let \( src_2 \) be the integer value in \( VR[VRB] \).

\( src_1 \) and \( src_2 \) can be signed or unsigned integers.

The rightmost 128 bits of the sum of \( src_1 \), the one's complement of \( src_2 \), and the value 1 are placed into \( VR[VRT] \).

**Special Registers Altered:**
None

### Vector Subtract Extended Unsigned Quadword Modulo VA-form

<table>
<thead>
<tr>
<th>vsubeuqm</th>
<th>VRT, VRA, VRB, VRC</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>6</td>
</tr>
</tbody>
</table>

```plaintext
if MSR.VEC=0 then Vector_Unavailable();
src1 ← VR[VRA];
src2 ← VR[VRB];
cin ← VR[VRC].bit[127];
sum ← EXTZ(src1) + EXTZ(~src2) + EXTZ(cin);
VR[VRT] ← Chop(sum, 128);
```

Let \( src_1 \) be the integer value in \( VR[VRA] \).
Let \( src_2 \) be the integer value in \( VR[VRB] \).
Let \( cin \) be the integer value in bit 127 of \( VR[VRC] \).

\( src_1 \) and \( src_2 \) can be signed or unsigned integers.

The rightmost 128 bits of the sum of \( src_1 \), the one's complement of \( src_2 \), and \( cin \) are placed into \( VR[VRT] \).

**Special Registers Altered:**
None

### Vector Subtract & write Carry Unsigned Quadword VX-form

<table>
<thead>
<tr>
<th>vsubcuq</th>
<th>VRT, VRA, VRB</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>6</td>
</tr>
</tbody>
</table>

```plaintext
if MSR.VEC=0 then Vector_Unavailable();
src1 ← VR[VRA];
src2 ← VR[VRB];
sum ← EXTZ(src1) + EXTZ(~src2) + EXTZ(1);
VR[VRT] ← Chop( EXTZ( Chop(sum >> 128, 1) ), 128 );
```

Let \( src_1 \) be the integer value in \( VR[VRA] \).
Let \( src_2 \) be the integer value in \( VR[VRB] \).

\( src_1 \) and \( src_2 \) can be signed or unsigned integers.

The carry out of the sum of \( src_1 \), the one's complement of \( src_2 \), and the value 1 is placed into \( VR[VRT] \).

**Special Registers Altered:**
None

### Vector Subtract Extended & write Carry Unsigned Quadword VA-form

<table>
<thead>
<tr>
<th>vsubcuq</th>
<th>VRT, VRA, VRB, VRC</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>6</td>
</tr>
</tbody>
</table>

```plaintext
if MSR.VEC=0 then Vector_Unavailable();
src1 ← VR[VRA];
src2 ← VR[VRB];
cin ← VR[VRC].bit[127];
sum ← EXTZ(src1) + EXTZ(~src2) + EXTZ(cin);
VR[VRT] ← Chop( EXTZ( Chop(sum >> 128, 1) ), 128 );
```

Let \( src_1 \) be the integer value in \( VR[VRA] \).
Let \( src_2 \) be the integer value in \( VR[VRB] \).
Let \( cin \) be the integer value in bit 127 of \( VR[VRC] \).

\( src_1 \) and \( src_2 \) can be signed or unsigned integers.

The carry out of the sum of \( src_1 \), the one's complement of \( src_2 \), and \( cin \) are placed into \( VR[VRT] \).

**Special Registers Altered:**
None
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.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>vsubuqm</code> vS3,vA3,vB3</td>
<td># bits 384:511 of difference</td>
</tr>
<tr>
<td><code>vsubcuq</code> vC3,vA3,vB3</td>
<td># carry out of bit 384 of difference</td>
</tr>
<tr>
<td><code>vsubuqm</code> vS2,vA2,vB2,vC3</td>
<td># bits 256:383 of difference</td>
</tr>
<tr>
<td><code>vsubcuq</code> vC2,vA2,vB2,vC3</td>
<td># carry out of bit 256 of difference</td>
</tr>
<tr>
<td><code>vsubuqm</code> vS1,vA1,vB1,vC2</td>
<td># bits 128:255 of difference</td>
</tr>
<tr>
<td><code>vsubcuq</code> vC1,vA1,vB1,vC2</td>
<td># carry out of bit 128 of difference</td>
</tr>
<tr>
<td><code>vsubuqm</code> vS0,vA0,vB0,vC1</td>
<td># bits 0:127 of difference</td>
</tr>
</tbody>
</table>
6.9.1.3 Vector Integer Multiply Instructions

**Vector Multiply Even Signed Byte VX-form**

\[
\text{vmulesb} \quad \text{VRT, VRA, VRB}
\]

```
4 6 11 16 21 776
```

```
do i=0 to 127 by 16
    prod ← EXTS((VRA)_{i:i+7}) \times \text{si} \ EXTS((VRB)_{i:i+7})
    \text{VRT}_{i:i+15} ← \text{Chop}(\text{prod}, 16)
end
```

For each integer value \( i \) from 0 to 7, do the following.
- Signed-integer byte element \( i \times 2 \) in VRA is multiplied by signed-integer byte element \( i \times 2 \) in VRB. The low-order 16 bits of the product are placed into halfword element \( i \) VRT.

Special Registers Altered:
- None

**Vector Multiply Odd Signed Byte VX-form**

\[
\text{vmulosb} \quad \text{VRT, VRA, VRB}
\]

```
4
```

```
0 6 11 16 21 264
```

```
do i=0 to 127 by 16
    prod ← EXTS((VRA)_{i+8:i+15}) \times \text{si} \ EXTS((VRB)_{i+8:i+15})
    \text{VRT}_{i:i+15} ← \text{Chop}(\text{prod}, 16)
end
```

For each integer value \( i \) from 0 to 7, do the following.
- Signed-integer byte element \( i \times 2 + 1 \) in VRA is multiplied by signed-integer byte element \( i \times 2 + 1 \) in VRB. The low-order 16 bits of the product are placed into halfword element \( i \) VRT.

Special Registers Altered:
- None

**Vector Multiply Even Unsigned Byte VX-form**

\[
\text{vmuleub} \quad \text{VRT, VRA, VRB}
\]

```
4
```

```
0 6 11 16 21 520
```

```
do i=0 to 127 by 16
    prod ← EXTS((VRA)_{i:i+7}) \times \text{ui} \ EXTS((VRB)_{i:i+7})
    \text{VRT}_{i:i+15} ← \text{Chop}(\text{prod}, 16)
end
```

For each integer value \( i \) from 0 to 7, do the following.
- Unsigned-integer byte element \( i \times 2 \) in VRA is multiplied by unsigned-integer byte element \( i \times 2 \) in VRB. The low-order 16 bits of the product are placed into halfword element \( i \) VRT.

Special Registers Altered:
- None

**Vector Multiply Odd Unsigned Byte VX-form**

\[
\text{vmuloub} \quad \text{VRT, VRA, VRB}
\]

```
4
```

```
0 6 11 16 21 8
```

```
do i=0 to 127 by 16
    prod ← EXTZ((VRA)_{i+8:i+15}) \times \text{ui} \ EXTZ((VRB)_{i+8:i+15})
    \text{VRT}_{i:i+15} ← \text{Chop}(\text{prod}, 16)
end
```

For each integer value \( i \) from 0 to 7, do the following.
- Unsigned-integer byte element \( i \times 2 + 1 \) in VRA is multiplied by unsigned-integer byte element \( i \times 2 + 1 \) in VRB. The low-order 16 bits of the product are placed into halfword element \( i \) VRT.

Special Registers Altered:
- None
Vector Multiply Even Signed Halfword
VX-form

\[ \text{vmulesh VRT,VRA,VRB} \]

\[
do \ i=0 \ to \ 127 \ by \ 32 \\
\quad \text{prod} \leftarrow \text{EXTS}((\text{VRA}_{i:i+15}) \times \text{EXTS}((\text{VRB}_{i:i+15})) \\
\quad \text{VRT}_{i:i+31} \leftarrow \text{Chop} \left( \text{prod}, \ 32 \right) \\
\end{array}
\]

For each integer value i from 0 to 3, do the following.
Signed-integer halfword element \( i \times 2 \) in VRA is multiplied by signed-integer halfword element \( i \times 2 \) in VRB. The low-order 32 bits of the product are placed into halfword element \( i \) VRT.

Special Registers Altered:
None

Vector Multiply Odd Signed Halfword
VX-form

\[ \text{vmulosh VRT,VRA,VRB} \]

\[
do \ i=0 \ to \ 127 \ by \ 32 \\
\quad \text{prod} \leftarrow \text{EXTS}((\text{VRA}_{i+16:i+31}) \times \text{EXTS}((\text{VRB}_{i+16:i+31})) \\
\quad \text{VRT}_{i:i+31} \leftarrow \text{Chop} \left( \text{prod}, \ 32 \right) \\
\end{array}
\]

For each integer value i from 0 to 3, do the following.
Signed-integer halfword element \( i \times 2+1 \) in VRA is multiplied by signed-integer halfword element \( i \times 2+1 \) in VRB. The low-order 32 bits of the product are placed into halfword element \( i \) VRT.

Special Registers Altered:
None

Vector Multiply Even Unsigned Halfword
VX-form

\[ \text{vmuleuh VRT,VRA,VRB} \]

\[
do \ i=0 \ to \ 127 \ by \ 32 \\
\quad \text{prod} \leftarrow \text{EXTZ}((\text{VRA}_{i:i+15}) \times \text{EXTZ}((\text{VRB}_{i:i+15})) \\
\quad \text{VRT}_{i:i+31} \leftarrow \text{Chop} \left( \text{prod}, \ 32 \right) \\
\end{array}
\]

For each integer value i from 0 to 3, do the following.
Unsigned-integer halfword element \( i \times 2 \) in VRA is multiplied by unsigned-integer halfword element \( i \times 2 \) in VRB. The low-order 32 bits of the product are placed into halfword element \( i \) VRT.

Special Registers Altered:
None

Vector Multiply Odd Unsigned Halfword
VX-form

\[ \text{vmulouh VRT,VRA,VRB} \]

\[
do \ i=0 \ to \ 127 \ by \ 32 \\
\quad \text{prod} \leftarrow \text{EXTZ}((\text{VRA}_{i+16:i+31}) \times \text{EXTZ}((\text{VRB}_{i+16:i+31})) \\
\quad \text{VRT}_{i:i+31} \leftarrow \text{Chop} \left( \text{prod}, \ 32 \right) \\
\end{array}
\]

For each integer value i from 0 to 3, do the following.
Unsigned-integer halfword element \( i \times 2+1 \) in VRA is multiplied by unsigned-integer halfword element \( i \times 2+1 \) in VRB. The low-order 32 bits of the product are placed into halfword element \( i \) VRT.

Special Registers Altered:
None
**Vector Multiply Even Signed Word**

**VX-form**

vmulesw VRT,VRA,VRB

```
<table>
<thead>
<tr>
<th></th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
</tr>
</thead>
<tbody>
<tr>
<td>VRT</td>
<td>VRA</td>
<td>VRB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
do i = 0 to 1
  src1 ← VR[VRA].word[2×i]
  src2 ← VR[VRB].word[2×i]
  VR[VRT].dword[i] ← src1 ×si src2
end
```

For each integer value \( i \) from 0 to 1, do the following.

The signed integer in word element \( 2 \times i \) of VR[VRA] is multiplied by the signed integer in word element \( 2 \times i \) of VR[VRB].

The 64-bit product is placed into doubleword element \( i \) of VR[VRT].

Special Registers Altered:
None

**Vector Multiply Odd Signed Word**

**VX-form**

vmulosw VRT,VRA,VRB

```
<table>
<thead>
<tr>
<th></th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
</tr>
</thead>
<tbody>
<tr>
<td>VRT</td>
<td>VRA</td>
<td>VRB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
do i = 0 to 1
  src1 ← VR[VRA].word[2×i+1]
  src2 ← VR[VRB].word[2×i+1]
  VR[VRT].dword[i] ← src1 ×si src2
end
```

For each integer value \( i \) from 0 to 1, do the following.

The signed integer in word element \( 2 \times i +1 \) of VR[VRA] is multiplied by the signed integer in word element \( 2 \times i +1 \) of VR[VRB].

The 64-bit product is placed into doubleword element \( i \) of VR[VRT].

Special Registers Altered:
None

**Vector Multiply Even Unsigned Word**

**VX-form**

vmuleuw VRT,VRA,VRB

```
<table>
<thead>
<tr>
<th></th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
</tr>
</thead>
<tbody>
<tr>
<td>VRT</td>
<td>VRA</td>
<td>VRB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
do i = 0 to 1
  src1 ← VR[VRA].word[2×i]
  src2 ← VR[VRB].word[2×i]
  VR[VRT].dword[i] ← src1 ×ui src2
end
```

For each integer value \( i \) from 0 to 1, do the following.

The unsigned integer in word element \( 2 \times i \) of VR[VRA] is multiplied by the unsigned integer in word element \( 2 \times i \) of VR[VRB].

The 64-bit product is placed into doubleword element \( i \) of VR[VRT].

Special Registers Altered:
None

**Vector Multiply Odd Unsigned Word**

**VX-form**

vmulouw VRT,VRA,VRB

```
<table>
<thead>
<tr>
<th></th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
</tr>
</thead>
<tbody>
<tr>
<td>VRT</td>
<td>VRA</td>
<td>VRB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
do i = 0 to 1
  src1 ← VR[VRA].word[2×i+1]
  src2 ← VR[VRB].word[2×i+1]
  VR[VRT].dword[i] ← src1 ×ui 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 VR[VRA] is multiplied by the unsigned integer in word element \( 2 \times i +1 \) of VR[VRB].

The 64-bit product is placed into doubleword element \( i \) of VR[VRT].

Special Registers Altered:
None
Vector Multiply Unsigned Word Modulo VX-form

vmuluwm VRT, VRA, VRB

\[
\begin{array}{cccccc}
4 & & & & & 137 \\
0 & 6 & 11 & 16 & 21 & 31 \\
\end{array}
\]

\[
do \ i = 0 \ to \ 3 \\
\quad \ src1 \gets \ VR[VRA].word[i] \\
\quad \ src2 \gets \ VR[VRB].word[i] \\
\quad \ VR[VRT].word[i] \gets \text{Chop}(src1 \times_{ui} src2, 32)
\end{do}

The integer in word element \(i\) of \(VR[VRA]\) is multiplied by the integer in word element \(i\) of \(VR[VRB]\).

The least-significant 32 bits of the product are placed into word element \(i\) of \(VR[VRT]\).

Special Registers Altered:
None

Programming Note

\textit{vmuluwm} can be used for unsigned or signed integers.
6.9.1.4 Vector Integer Multiply-Add/Sum Instructions

**Vector Multiply-High-Add Signed Halfword Saturate VA-form**

\[ \text{vmhaddshs VRT, VRA, VRB, VRC} \]

```
\begin{array}{|c|c|c|c|c|c|}
\hline
4 & 6 & 11 & 16 & 21 & 26 & 31 \\
\hline
\end{array}
```

do \text{i} = 0 to 127 by 16
prod ≜ \text{EXTS}((\text{VRA}_i:i+15)) \times_{\text{si}} \text{EXTS}((\text{VRB}_i:i+15))
sum ≜ (\text{prod} >>_{\text{si}} 15) +_{\text{int}} \text{EXTS}((\text{VRC}_i:i+15))
VRT_{i:i+15} ≜ \text{Clamp}(\text{sum}, -2^{15}, 2^{15} - 1)
end

For each vector element \text{i} from 0 to 7, do the following.
Signed-integer halfword element \text{i} in VRA is multiplied by signed-integer halfword element \text{i} in VRB, producing a 32-bit signed-integer product. Bits 0:16 of the product are added to signed-integer halfword element \text{i} in VRC.

- If the intermediate result is greater than \(2^{15} - 1\), the result saturates to \(2^{15} - 1\).
- If the intermediate result is less than \(-2^{15}\), the result saturates to \(-2^{15}\).

The low-order 16 bits of the result are placed into halfword element \text{i} of VRT.

**Special Registers Altered:**
SAT

**Vector Multiply-High-Round-Add Signed Halfword Saturate VA-form**

\[ \text{vmhraddshs VRT, VRA, VRB, VRC} \]

```
\begin{array}{|c|c|c|c|c|c|}
\hline
4 & 6 & 11 & 16 & 21 & 26 & 31 \\
\hline
\end{array}
```

do \text{i} = 0 to 127 by 16
\text{temp} ≜ \text{EXTS}((\text{VRC}_i:i+15))
prod ≜ \text{EXTS}((\text{VRA}_i:i+15)) \times_{\text{si}} \text{EXTS}((\text{VRB}_i:i+15))
sum ≜ (\text{prod} +_{\text{int}} 0x0000_4000) >>_{\text{si}} 15 +_{\text{int}} \text{temp}
VRT_{i:i+15} ≜ \text{Clamp}(\text{sum}, -2^{15}, 2^{15} - 1)
end

For each vector element \text{i} from 0 to 7, do the following.
Signed-integer halfword element \text{i} in VRA is multiplied by signed-integer halfword element \text{i} in VRB, producing a 32-bit signed-integer product. The value \(0x0000_4000\) is added to the product, producing a 32-bit signed-integer sum. Bits 0:16 of the sum are added to signed-integer halfword element \text{i} in VRC.

- If the intermediate result is greater than \(2^{15} - 1\), the result saturates to \(2^{15} - 1\).
- If the intermediate result is less than \(-2^{15}\), the result saturates to \(-2^{15}\).

The low-order 16 bits of the result are placed into halfword element \text{i} of VRT.

**Special Registers Altered:**
SAT
Vector Multiply-Low-Add Unsigned Halfword Modulo VA-form

**vmladduhm** VRT, VRA, VRB, VRC

<table>
<thead>
<tr>
<th></th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>VRC</th>
<th>34</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>

\[ \text{do } i = 0 \text{ to } 127 \text{ by } 16 \]

\[ \text{prod } \leftarrow \text{EXTZ}(VRA)_{i:i+15} \times \text{EXTZ}(VRB)_{i:i+15} \]

\[ \text{sum } \leftarrow \text{Chop}(\text{prod}, 16) + \text{int}(VRC)_{i:i+15} \]

\[ \text{VRT}_{i:i+15} \leftarrow \text{Chop}(\text{sum}, 16) \]

For each integer value \( i \) from 0 to 3, do the following.

Unsigned-integer halfword element \( i \) in VRA is multiplied by unsigned-integer halfword element \( i \) in VRB, producing a 32-bit unsigned-integer product. The low-order 16 bits of the product are added to unsigned-integer halfword element \( i \) in VRC.

The low-order 16 bits of the sum are placed into halfword element \( i \) of VRT.

**Special Registers Altered:**

None

**Programming Note**

\textit{vmladduhm} can be used for unsigned or signed-integers.

Vector Multiply-Sum Unsigned Byte Modulo VA-form

**vmsumubm** VRT, VRA, VRB, VRC

<table>
<thead>
<tr>
<th></th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>VRC</th>
<th>36</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>

\[ \text{do } i = 0 \text{ to } 127 \text{ by } 32 \]

\[ \text{temp } \leftarrow \text{EXTZ}(VRC)_{i:i+31} \]

\[ \text{do } j = 0 \text{ to } 31 \text{ by } 8 \]

\[ \text{prod } \leftarrow \text{EXTZ}(VRA)_{i+j:i+j+7} \times \text{EXTZ}(VRB)_{i+j:i+j+7} \]

\[ \text{temp } \leftarrow \text{temp } + \text{int prod} \]

\[ \text{VRT}_{i:i+31} \leftarrow \text{Chop}(\text{temp}, 32) \]

For each word element in VRT the following operations are performed, in the order shown.

- Each of the four unsigned-integer byte elements contained in the corresponding word element of VRA is multiplied by the corresponding unsigned-integer byte element in VRB, producing an unsigned-integer halfword product.

- The sum of these four unsigned-integer halfword products is added to the unsigned-integer word element in VRC.

- The unsigned-integer word result is placed into the corresponding word element of VRT.

**Special Registers Altered:**

None
Vector Multiply-Sum Mixed Byte Modulo VA-form

$vmsummbm$ VRT, VRA, VRB, VRC

<table>
<thead>
<tr>
<th>4</th>
<th>6</th>
<th>VRT</th>
<th>11</th>
<th>16</th>
<th>VRB</th>
<th>21</th>
<th>26</th>
<th>31</th>
</tr>
</thead>
</table>

do $i = 0$ to $127$ by $12$
  temp $\leftarrow (VRC)_{i:i+31}$
  do $j = 0$ to $31$ by $8$
    prod0:15 $\leftarrow (VRA)_{i+j:i+j+7} \times^s (VRB)_{i+j:i+j+7}$
    temp $\leftarrow temp \cdot int\ EXTS(prod)$
  end
  VRT$_{i:i+31} \leftarrow temp$
end

For each word element in VRT the following operations are performed, in the order shown.

- Each of the four signed-integer byte elements contained in the corresponding word element of VRA is multiplied by the corresponding unsigned-integer byte element in VRB, producing a signed-integer product.

- The sum of these four signed-integer halfword products is added to the signed-integer word element in VRC.

- The signed-integer result is placed into the corresponding word element of VRT.

Special Registers Altered:
None

Vector Multiply-Sum Signed Halfword Modulo VA-form

$vmsumshm$ VRT, VRA, VRB, VRC

<table>
<thead>
<tr>
<th>4</th>
<th>6</th>
<th>VRT</th>
<th>11</th>
<th>16</th>
<th>VRB</th>
<th>21</th>
<th>26</th>
<th>31</th>
</tr>
</thead>
</table>

do $i = 0$ to $127$ by $12$
  temp $\leftarrow (VRC)_{i:i+31}$
  do $j = 0$ to $31$ by $16$
    prod0:31 $\leftarrow (VRA)_{i+j:i+j+15} \times^s (VRB)_{i+j:i+j+15}$
    temp $\leftarrow temp \cdot int\ Prod$
  end
  VRT$_{i:i+31} \leftarrow temp$
end

For each word element in VRT the following operations are performed, in the order shown.

- Each of the two signed-integer halfword elements contained in the corresponding word element of VRA is multiplied by the corresponding signed-integer halfword element in VRB, producing a signed-integer product.

- The sum of these two signed-integer word products is added to the signed-integer word element in VRC.

- The signed-integer word result is placed into the corresponding word element of VRT.

Special Registers Altered:
None
**Vector Multiply-Sum Signed Halfword Saturate VA-form**

\[ \text{vmsumshs} \quad \text{VRT, VRA, VRB, VRC} \]

\[
\begin{array}{cccccc}
0 & 4 & 6 & 11 & 16 & 21 & 26 & 31 & 41 \\
\end{array}
\]

\[
\begin{aligned}
do i=0 \text{ to } 127 \text{ by } 32 \\
\text{temp} & \leftarrow \text{EXTS}((\text{VRC})_{i:i+31}) \\
do j=0 \text{ to } 31 \text{ by } 16 \\
\text{srcA} & \leftarrow \text{EXTS}((\text{VRA})_{i+j:i+j+15}) \\
\text{srcB} & \leftarrow \text{EXTS}((\text{VRB})_{i+j:i+j+15}) \\
\text{prod} & \leftarrow \text{srcA} \times_{\text{si}} \text{srcB} \\
\text{temp} & \leftarrow \text{temp} +_{\text{int}} \text{prod} \\
\text{VRT}_{i:i+31} & \leftarrow \text{Clamp}((\text{temp}, -2^{31}, 2^{31}-1) \\
\end{aligned}
\]

For each word element in VRT the following operations are performed, in the order shown.

- Each of the two signed-integer halfword elements contained in the corresponding word element of VRA is multiplied by the corresponding signed-integer halfword element in VRB, producing a signed-integer product.

- The sum of these two signed-integer word products is added to the signed-integer word element in VRC.

- If the intermediate result is greater than \(2^{31}-1\) the result saturates to \(2^{31}-1\) and if it is less than \(-2^{31}\) it saturates to \(-2^{31}\).

- The result is placed into the corresponding word element of VRT.

**Special Registers Altered:**

SAT

---

**Vector Multiply-Sum Unsigned Halfword Modulo VA-form**

\[ \text{vmsuhtm} \quad \text{VRT, VRA, VRB, VRC} \]

\[
\begin{array}{cccccc}
0 & 4 & 6 & 11 & 16 & 21 & 26 & 31 & 38 \\
\end{array}
\]

\[
\begin{aligned}
do i=0 \text{ to } 127 \text{ by } 32 \\
\text{temp} & \leftarrow \text{EXTZ}((\text{VRC})_{i:i+31}) \\
do j=0 \text{ to } 31 \text{ by } 16 \\
\text{srcA} & \leftarrow \text{EXTZ}((\text{VRA})_{i+j:i+j+15}) \\
\text{srcB} & \leftarrow \text{EXTZ}((\text{VRB})_{i+j:i+j+15}) \\
\text{prod} & \leftarrow \text{srcA} \times_{ui} \text{srcB} \\
\text{temp} & \leftarrow \text{temp} +_{\text{int}} \text{prod} \\
\text{VRT}_{i:i+31} & \leftarrow \text{Chop}(\text{temp}, 32) \\
\end{aligned}
\]

For each word element in VRT the following operations are performed, in the order shown.

- Each of the two unsigned-integer halfword elements contained in the corresponding word element of VRA is multiplied by the corresponding unsigned-integer halfword element in VRB, producing an unsigned-integer word product.

- The sum of these two unsigned-integer word products is added to the unsigned-integer word element in VRC.

- The unsigned-integer result is placed into the corresponding word element of VRT.

**Special Registers Altered:**

None
Vector Multiply-Sum Unsigned Halfword Saturate VA-form

vmsumuhs 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>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>26</td>
</tr>
</tbody>
</table>

do i=0 to 127 by 32
  temp = EXTZ((VRC)i:i+31)
do j=0 to 31 by 16
  src1 = EXTZ((VRA)i+j:i+j+15)
  src2 = EXTZ((VRB)i+j:i+j+15)
  prod = src1 ×ui src2
  temp = temp + prod
end
VRTi:i+31 = Clamp(temp, 0, 2^32-1)

For each word element in VRT the following operations are performed, in the order shown.

- Each of the two unsigned-integer halfword elements contained in the corresponding word element of VRA is multiplied by the corresponding unsigned-integer halfword element in VRB, producing an unsigned-integer product.

- The sum of these two unsigned-integer word products is added to the unsigned-integer word element in VRC.

- If the intermediate result is greater than 2^32-1 the result saturates to 2^32-1.

- The result is placed into the corresponding word element of VRT.

Special Registers Altered:
SAT

Vector Multiply-Sum Unsigned Doubleword Modulo VA-form

vmsumudm 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>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>26</td>
</tr>
</tbody>
</table>

temp = EXTZ(VR[VRC])
do i = 0 to 1
  prod = EXTZ(VR[VRA].dword[i]) × EXTZ(VR[VRB].dword[i])
  temp = temp + prod
end
VRT[VRT] = Chop(temp, 128)

The unsigned integer value in doubleword element 0 of VR[VRA] is multiplied by the unsigned integer value in doubleword element 0 of VR[VRB] to produce a 128-bit product.

The unsigned integer value in doubleword element 1 of VR[VRA] is multiplied by the unsigned integer value in doubleword element 1 of VR[VRB] to produce a 128-bit product.

The two 128-bit unsigned integer products and the 128-bit unsigned integer in VR[VRC] are summed.

The low-order 128 bits of the sum are placed into VR[VRT]. Any carry out or overflow status is discarded.

Special Registers Altered:
None

**Programming Note**

A horizontal add of the doubleword elements in VR[VRA] can be performed using vmsumudm when VR[VRB] contains the doubleword integer values \{1, 1\} and VR[VRC] contains the quadword integer value 0.

A horizontal subtract of the doubleword elements in VR[VRA] can be performed using vmsumudm when VR[VRB] contains the doubleword integer values \{1, -1\} and VR[VRC] contains the quadword integer value 0.

A multiply even unsigned doubleword operation can be performed using vmsumudm when the contents of doubleword element 1 of VR[VRA] or VR[VRB] are 0 and the contents of VR[VRC] to 0.

A multiply odd unsigned doubleword operation can be performed using vmsumudm when the contents of doubleword element 0 of VR[VRA] or VR[VRB] are 0 and the contents of VR[VRC] to 0.
6.9.1.5 Vector Integer Sum-Across Instructions

**Vector Sum across Signed Word Saturate VX-form**

\[ \text{vsumsws VRT, VRA, VRB} \]

<table>
<thead>
<tr>
<th>i</th>
<th>4</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>1928</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>

- \( \text{temp} \leftarrow \text{EXTS}((\text{VRB})_{96:127}) \)
- \( \text{do } i=0 \text{ to } 127 \text{ by } 32 \)
- \( \text{temp} \leftarrow \text{temp} \text{ +int EXTS}((\text{VRA})_{i:i+31}) \)
- \( \text{end} \)
- \( \text{VRT}_{0:31} \leftarrow 0x0000_0000 \)
- \( \text{VRT}_{32:63} \leftarrow 0x0000_0000 \)
- \( \text{VRT}_{64:95} \leftarrow 0x0000_0000 \)
- \( \text{VRT}_{96:127} \leftarrow \text{Clamp}(\text{temp}, -2^{31}, 2^{31}-1) \)

The sum of the four signed-integer word elements in VRA is added to signed-integer word element 3 of VRB.

- 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 low-end 32 bits of the result are placed into word element 3 of VRT.

Word elements 0 to 2 of VRT are set to 0.

**Special Registers Altered:**

SAT

---

**Vector Sum across Half Signed Word Saturate VX-form**

\[ \text{vsum2sws VRT, VRA, VRB} \]

<table>
<thead>
<tr>
<th>i</th>
<th>4</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>1672</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>31</td>
</tr>
</tbody>
</table>

- \( \text{do } i=0 \text{ to } 127 \text{ by } 64 \)
- \( \text{temp} \leftarrow \text{EXTS}((\text{VRB})_{i+32:i+63}) \)
- \( \text{do } j=0 \text{ to } 63 \text{ by } 32 \)
- \( \text{temp} \leftarrow \text{temp} \text{ +int EXTS}((\text{VRA})_{i+j:i+j+31}) \)
- \( \text{end} \)
- \( \text{VRT}_{i:i+63} \leftarrow 0x0000_0000 \| \text{Clamp}(\text{temp}, -2^{31}, 2^{31}-1) \)
- \( \text{end} \)

The sum of the signed-integer word elements 0 and 1 in VRA is added to the signed-integer word element in bits 32:63 of VRB.

- 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 low-order 32 bits of the result are placed into word element 1 of VRT.

The sum of signed-integer word elements 2 and 3 in VRA is added to the signed-integer word element in bits 96:127 of VRB.

- 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 low-order 32 bits of the result are placed into word element 3 of VRT.

**Special Registers Altered:**

SAT
Vector Sum across Quarter Signed Byte Saturate VX-form

vsum4sbs VRT, VRA, VRB

<table>
<thead>
<tr>
<th>4</th>
<th>VRB</th>
<th>VRA</th>
<th>VRT</th>
<th>1800</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

do i=0 to 127 by 32
  temp ← EXTS[(VRB)\text{i:i+31}]
  do j=0 to 31 by 8
    temp ← temp + \text{int} EXTS[(VRA)\text{i+j:i+j+7}]
  end
  VRT\text{i:i+31} ← Clamp(temp, -2^{31}, 2^{31}-1)
end

For each integer value i from 0 to 3, do the following.
The sum of the four signed-integer byte elements contained in word element i of VRA is added to signed-integer word element i in VRB.

- If the intermediate result is greater than $2^{31}$, the result saturates to $2^{31}-1$.
- If the intermediate result is less than $-2^{31}$, the result saturates to $-2^{31}$.

The low-order 32 bits of the result are placed into word element i of VRT.

Special Registers Altered:
SAT

Vector Sum across Quarter Signed Halfword Saturate VX-form

vsum4shs VRT, VRA, VRB

<table>
<thead>
<tr>
<th>4</th>
<th>VRB</th>
<th>VRA</th>
<th>VRT</th>
<th>1608</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

do i=0 to 127 by 32
  temp ← EXTS[(VRB)\text{i:i+31}]
  do j=0 to 31 by 16
    temp ← temp + \text{int} EXTS[(VRA)\text{i+j:i+j+15}]
  end
  VRT\text{i:i+31} ← Clamp(temp, -2^{31}, 2^{31}-1)
end

For each integer value i from 0 to 3, do the following.
The sum of the two signed-integer halfword elements contained in word element i of VRA is added to signed-integer word element i in VRB.

- If the intermediate result is greater than $2^{31}$, the result saturates to $2^{31}-1$.
- If the intermediate result is less than $-2^{31}$, the result saturates to $-2^{31}$.

The low-order 32 bits of the result are placed into the corresponding word element of VRT.

Special Registers Altered:
SAT
Vector Sum across Quarter Unsigned Byte Saturate VX-form

\texttt{vsum4ubs} \quad \texttt{VRT, VRA, VRB}

\begin{tabular}{cccccc}
\textbf{4} & \textbf{VRT} & \textbf{VRA} & \textbf{VRB} & \textbf{1544} \\
0 & 6 & 31 & 31 & 31 & 31 \\
\end{tabular}

\begin{verbatim}
do i=0 to 127 by 32
temp \leftarrow \text{EXTZ}((\text{VRB})_{i:i+31})
do j=0 to 31 by 8
temp \leftarrow temp + \text{EXTZ}((\text{VRA})_{i+j:i+j+7})
end
VRT_{i:i+31} \leftarrow \text{Clamp}(\text{temp}, 0, 2^{32}-1)
end
\end{verbatim}

For each integer value \( i \) from 0 to 3, do the following.
The sum of the four unsigned-integer byte elements contained in word element \( i \) of VRA is added to unsigned-integer word element \( i \) in VRB.

- If the intermediate result is greater than \( 2^{32} - 1 \) it saturates to \( 2^{32} - 1 \).

The low-order 32 bits of the result are placed into word element \( i \) of VRT.

Special Registers Altered:
\begin{itemize}
\item SAT
\end{itemize}
6.9.1.6 Vector Integer Negate Instructions

**Vector Negate Word VX-form**

\[
vnegw \quad VRT, VRB
\]

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 VR[VRB] and 1 is placed into word element \(i\) of VR[VRT].

**Special Registers Altered:**

None

**Vector Negate Doubleword VX-form**

\[
vnegd \quad VRT, VRB
\]

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 VR[VRB] and 1 is placed into doubleword element \(i\) of VR[VRT].

**Special Registers Altered:**

None
### 6.9.2 Vector Extend Sign Instructions

**Vector Extend Sign Byte To Word VX-form**

\[ \text{vextsb2w } \text{VRT,VRB} \]

<table>
<thead>
<tr>
<th>4</th>
<th>6</th>
<th>16</th>
<th>21</th>
<th>1538</th>
<th>31</th>
</tr>
</thead>
</table>

- If MSR.VEC=0 then Vector_Unavailable()
- \( \text{do } i = 0 \text{ to } 3 \)
  - \( \text{VR[VRT].word}[i] \leftarrow \text{EXTS32}(\text{VR[VRB].word}[i].\text{byte}[3]) \)
- \( \text{end} \)

For each integer value \( i \) from 0 to 3, do the following.

The rightmost byte of word element \( i \) of \( \text{VR[VRB]} \) is sign-extended and placed into word element \( i \) of \( \text{VR[VRT]} \).

**Special Registers Altered:**  None

**Vector Extend Sign Halfword To Word VX-form**

\[ \text{vextsh2w } \text{VRT,VRB} \]

<table>
<thead>
<tr>
<th>4</th>
<th>6</th>
<th>17</th>
<th>16</th>
<th>21</th>
<th>1538</th>
<th>31</th>
</tr>
</thead>
</table>

- If MSR.VEC=0 then Vector_Unavailable()
- \( \text{do } i = 0 \text{ to } 3 \)
  - \( \text{VR[VRT].word}[i] \leftarrow \text{EXTS32}(\text{VR[VRB].word}[i].\text{hword}[1]) \)
- \( \text{end} \)

For each integer value \( i \) from 0 to 3, do the following.

The rightmost halfword of word element \( i \) of \( \text{VR[VRB]} \) is sign-extended and placed into word element \( i \) of \( \text{VR[VRT]} \).

**Special Registers Altered:**  None

**Vector Extend Sign Word To Doubleword VX-form**

\[ \text{vextsw2d } \text{VRT,VRB} \]

<table>
<thead>
<tr>
<th>4</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>1538</th>
<th>31</th>
</tr>
</thead>
</table>

- If MSR.VEC=0 then Vector_Unavailable()
- \( \text{do } i = 0 \text{ to } 1 \)
  - \( \text{VR[VRT].dword}[i] \leftarrow \text{EXTS64}(\text{VR[VRB].dword}[i].\text{word}[1]) \)
- \( \text{end} \)

For each integer value \( i \) from 0 to 1, do the following.

The rightmost word of doubleword element \( i \) of \( \text{VR[VRB]} \) is sign-extended and placed into doubleword element \( i \) of \( \text{VR[VRT]} \).

**Special Registers Altered:**  None

**Vector Extend Sign Byte To Doubleword VX-form**

\[ \text{vextsb2d } \text{VRT,VRB} \]

<table>
<thead>
<tr>
<th>4</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>1538</th>
<th>31</th>
</tr>
</thead>
</table>

- If MSR.VEC=0 then Vector_Unavailable()
- \( \text{do } i = 0 \text{ to } 1 \)
  - \( \text{VR[VRT].dword}[i] \leftarrow \text{EXTS64}(\text{VR[VRB].dword}[i].\text{byte}[7]) \)
- \( \text{end} \)

For each integer value \( i \) from 0 to 1, do the following.

The rightmost byte of doubleword element \( i \) of \( \text{VR[VRB]} \) is sign-extended and placed into doubleword element \( i \) of \( \text{VR[VRT]} \).

**Special Registers Altered:**  None

**Vector Extend Sign Halfword To Doubleword VX-form**

\[ \text{vextsh2d } \text{VRT,VRB} \]

<table>
<thead>
<tr>
<th>4</th>
<th>6</th>
<th>25</th>
<th>16</th>
<th>21</th>
<th>1538</th>
<th>31</th>
</tr>
</thead>
</table>

- If MSR.VEC=0 then Vector_Unavailable()
- If “vextsh2d” then \( \text{do } i = 0 \text{ to } 1 \)
  - \( \text{VR[VRT].dword}[i] \leftarrow \text{EXTS64}(\text{VR[VRB].dword}[i].\text{hword}[3]) \)
- \( \text{end} \)

For each integer value \( i \) from 0 to 1, do the following.

The rightmost halfword of doubleword element \( i \) of \( \text{VR[VRB]} \) is sign-extended and placed into doubleword element \( i \) of \( \text{VR[VRT]} \).

**Special Registers Altered:**  None

**Vector Extend Sign Word To Doubleword VX-form**

\[ \text{vextsw2d } \text{VRT,VRB} \]

<table>
<thead>
<tr>
<th>4</th>
<th>6</th>
<th>28</th>
<th>16</th>
<th>21</th>
<th>1538</th>
<th>31</th>
</tr>
</thead>
</table>

- If MSR.VEC=0 then Vector_Unavailable()
- \( \text{do } i = 0 \text{ to } 1 \)
  - \( \text{VR[VRT].dword}[i] \leftarrow \text{EXTS64}(\text{VR[VRB].dword}[i].\text{word}[1]) \)
- \( \text{end} \)

For each integer value \( i \) from 0 to 1, do the following.

The rightmost word of doubleword element \( i \) of \( \text{VR[VRB]} \) is sign-extended and placed into doubleword element \( i \) of \( \text{VR[VRT]} \).

**Special Registers Altered:**  None
6.9.2.1 Vector Integer Average Instructions

**Vector Average Signed Byte VX-form**

\[ \text{vavgsb } \text{VRT, VRA, VRB} \]

<table>
<thead>
<tr>
<th></th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>1282</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td></td>
<td>6</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ \text{do } i=0 \text{ to } 127 \text{ by } 8 \]
\[ \text{aop } \gets \text{EXTS}((\text{VRA})_{i:i+7}) \]
\[ \text{bop } \gets \text{EXTS}((\text{VRB})_{i:i+7}) \]
\[ \text{VRT}_{i:i+7} \gets \text{Chop}(\text{aop } + \text{bop } + \text{int } 1 \gg 1, 8) \]

For each integer value \( i \) from 0 to 15, do the following. Signed-integer byte element \( i \) in VRA is added to signed-integer byte element \( i \) in VRB. 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 VRT.

Special Registers Altered:
None

**Vector Average Signed Halfword VX-form**

\[ \text{vavgsh } \text{VRT, VRA, VRB} \]

<table>
<thead>
<tr>
<th></th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>1346</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td></td>
<td>6</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ \text{do } i=0 \text{ to } 127 \text{ by } 16 \]
\[ \text{aop } \gets \text{EXTS}((\text{VRA})_{i:i+15}) \]
\[ \text{bop } \gets \text{EXTS}((\text{VRB})_{i:i+15}) \]
\[ \text{VRT}_{i:i+15} \gets \text{Chop}(\text{aop } + \text{bop } + \text{int } 1 \gg 1, 16) \]

For each integer value \( i \) from 0 to 7, do the following. Signed-integer halfword element \( i \) in VRA is added to signed-integer halfword element \( i \) in VRB. 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 VRT.

Special Registers Altered:
None

**Vector Average Signed Word VX-form**

\[ \text{vavgsw } \text{VRT, VRA, VRB} \]

<table>
<thead>
<tr>
<th></th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>1410</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td></td>
<td>6</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ \text{do } i=0 \text{ to } 127 \text{ by } 32 \]
\[ \text{aop } \gets \text{EXTS}((\text{VRA})_{i:i+31}) \]
\[ \text{bop } \gets \text{EXTS}((\text{VRB})_{i:i+31}) \]
\[ \text{VRT}_{i:i+31} \gets \text{Chop}(\text{aop } + \text{bop } + \text{int } 1 \gg 1, 32) \]

For each integer value \( i \) from 0 to 3, do the following. Signed-integer word element \( i \) in VRA is added to signed-integer word element \( i \) in VRB. 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 VRT.

Special Registers Altered:
None
Set register VRT (VT) to the vector average of the bytes of registers VRA (VA) and VRB (VB), where each is an n-bit word.

\[
\text{VX-form: } \text{vavgub } \text{VRT, VRA, VRB}
\]

<table>
<thead>
<tr>
<th>i</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>6</td>
<td>11</td>
<td>1026</td>
</tr>
</tbody>
</table>

For each integer value \( i \) from 0 to 15, do the following.

- Set \( \text{aop} \leftarrow \text{EXTZ} ((\text{VRA})_{i:i+7}) \)
- Set \( \text{bop} \leftarrow \text{EXTZ} ((\text{VRB})_{i:i+7}) \)
- Set \( \text{VRT}_{i:i+7} \leftarrow \text{Chop} ((\text{aop} + \text{int bop} + \text{int 1}) \gg \text{ui} 1, 8) \)

The low-order 8 bits of the result are placed into byte element \( i \) of VRT.

Special Registers Altered:
None

Set register VRT (VT) to the vector average of the halfwords of registers VRA (VA) and VRB (VB), where each is an n-bit word.

\[
\text{VX-form: } \text{vavguh } \text{VRT, VRA, VRB}
\]

<table>
<thead>
<tr>
<th>i</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>6</td>
<td>16</td>
<td>1090</td>
</tr>
</tbody>
</table>

For each integer value \( i \) from 0 to 7, do the following.

- Set \( \text{aop} \leftarrow \text{EXTZ} ((\text{VRA})_{i:i+15}) \)
- Set \( \text{bop} \leftarrow \text{EXTZ} ((\text{VRB})_{i:i+15}) \)
- Set \( \text{VRT}_{i:i+15} \leftarrow \text{Chop} ((\text{aop} + \text{int bop} + \text{int 1}) \gg \text{ui} 1, 16) \)

The low-order 16 bits of the result are placed into halfword element \( i \) of VRT.

Special Registers Altered:
None

Set register VRT (VT) to the vector average of the words of registers VRA (VA) and VRB (VB), where each is an n-bit word.

\[
\text{VX-form: } \text{vavguw } \text{VRT, VRA, VRB}
\]

<table>
<thead>
<tr>
<th>i</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>6</td>
<td>21</td>
<td>1154</td>
</tr>
</tbody>
</table>

For each integer value \( i \) from 0 to 3, do the following.

- Set \( \text{aop} \leftarrow \text{EXTZ} ((\text{VRA})_{i:i+31}) \)
- Set \( \text{bop} \leftarrow \text{EXTZ} ((\text{VRB})_{i:i+31}) \)
- Set \( \text{VRT}_{i:i+31} \leftarrow \text{Chop} ((\text{aop} + \text{int bop} + \text{int 1}) \gg \text{ui} 1, 32) \)

The low-order 32 bits of the result are placed into word element \( i \) of VRT.

Special Registers Altered:
None
### 6.9.2.2 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 Signed Byte

**VX-form**

\[
\text{vabsdb} \text{ VRT}, \text{VRA}, \text{VRB}
\]

<table>
<thead>
<tr>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>1026</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>6</td>
<td>11</td>
<td>16</td>
</tr>
</tbody>
</table>

**Vraman**

\[
\text{vabsdb} \text{ VRT}, \text{VRA}, \text{VRB}
\]

**Vxraman**

\[
\text{vabsdb} \text{ VRT}, \text{VRA}, \text{VRB}
\]

**Vrman**

\[
\text{vabsdb} \text{ VRT}, \text{VRA}, \text{VRB}
\]

**Vxprman**

\[
\text{vabsdb} \text{ VRT}, \text{VRA}, \text{VRB}
\]

For each integer value \(i\) from 0 to 15, do the following.

The unsigned integer value in byte element \(i\) of \(\text{VR}[\text{VRA}]\) is subtracted by the unsigned integer value in byte element \(i\) of \(\text{VR}[\text{VRB}]\). The absolute value of the difference is placed into byte element \(i\) of \(\text{VR}[\text{VRT}]\).

**Special Registers Altered:**

None

#### Vector Absolute Difference Signed Halfword

**VX-form**

\[
\text{vabsdub} \text{ VRT}, \text{VRA}, \text{VRB}
\]

<table>
<thead>
<tr>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>1090</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
</tr>
</tbody>
</table>

**Vraman**

\[
\text{vabsdub} \text{ VRT}, \text{VRA}, \text{VRB}
\]

**Vxraman**

\[
\text{vabsdub} \text{ VRT}, \text{VRA}, \text{VRB}
\]

**Vrman**

\[
\text{vabsdub} \text{ VRT}, \text{VRA}, \text{VRB}
\]

**Vxprman**

\[
\text{vabsdub} \text{ VRT}, \text{VRA}, \text{VRB}
\]

For each integer value \(i\) from 0 to 7, do the following.

The unsigned integer value in halfword element \(i\) of \(\text{VR}[\text{VRA}]\) is subtracted by the unsigned integer value in halfword element \(i\) of \(\text{VR}[\text{VRB}]\). The absolute value of the difference is placed into halfword element \(i\) of \(\text{VR}[\text{VRT}]\).

**Special Registers Altered:**

None
Vector Absolute Difference Unsigned Word VX-form

vabsduw VRT,VRA,VRB

<table>
<thead>
<tr>
<th></th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>1155</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 = 0 to 3
  src1 \(\leftarrow\) EXTZ(VR[VRA].word[i])
  src2 \(\leftarrow\) EXTZ(VR[VRB].word[i])
  if (src1>src2) then
    VR[VRT].word[i] \(\leftarrow\) Chop(src1 + ¬src2 + 1, 32)
  else
    VR[VRT].word[i] \(\leftarrow\) Chop(src2 + ¬src1 + 1, 32)
end

For each integer value i from 0 to 3, do the following.
The unsigned integer value in word element i of VR[VRA] is subtracted by the unsigned integer value in word element i of VR[VRB]. The absolute value of the difference is placed into word element i of VR[VRT].

Special Registers Altered:
None
6.9.2.3 Vector Integer Maximum and Minimum Instructions

**Vector Maximum Signed Byte VX-form**

\[ \text{vmaxsb} \quad \text{VRT, VRA, VRB} \]

\[
\begin{array}{cccc}
4 & 6 & 11 & 16 & 21 & 258 \\
\end{array}
\]

\[
\begin{array}{c}
do \ i=0 \ to \ 127 \ by \ 8 \\
\quad \text{aop} \leftarrow \text{EXTS}((\text{VRA})_{i:i+7}) \\
\quad \text{bop} \leftarrow \text{EXTS}((\text{VRB})_{i:i+7}) \\
\quad \text{VRT}_{i:i+7} \leftarrow \text{if } (\text{aop} > \text{bop}) \text{ then } (\text{VRA})_{i:i+7} \text{ else } (\text{VRB})_{i:i+7} \\
\end{array}
\]

For each integer value \( i \) from 0 to 15, do the following.
Signed-integer byte element \( i \) in VRA is compared to signed-integer byte element \( i \) in VRB. The larger of the two values is placed into byte element \( i \) of VRT.

Special Registers Altered:
None

**Vector Maximum Signed Doubleword VX-form**

\[ \text{vmaxsd} \quad \text{VRT, VRA, VRB} \]

\[
\begin{array}{cccc}
4 & 6 & 11 & 16 & 21 & 450 \\
\end{array}
\]

\[
\begin{array}{c}
do \ i=0 \ to \ 1 \\
\quad \text{aop} \leftarrow \text{VR}[\text{VRA}].\text{dword}[i] \\
\quad \text{bop} \leftarrow \text{VR}[\text{VRB}].\text{dword}[i] \\
\quad \text{VR}[\text{VRT}].\text{dword}[i] \leftarrow \text{if } (\text{aop} > \text{bop}) \text{ then } \text{aop} \text{ else } \text{bop} \\
\end{array}
\]

For each integer value \( i \) from 0 to 1, do the following.
The signed integer value in doubleword element \( i \) of VR[VRT] is compared to the signed integer value in doubleword element \( i \) of VR[VRB]. The larger of the two values is placed into doubleword element \( i \) of VR[VRT].

Special Registers Altered:
None

**Vector Maximum Unsigned Byte VX-form**

\[ \text{vmaxub} \quad \text{VRT, VRA, VRB} \]

\[
\begin{array}{cccc}
4 & 6 & 11 & 16 & 21 & 2 \\
\end{array}
\]

\[
\begin{array}{c}
do \ i=0 \ to \ 127 \ by \ 8 \\
\quad \text{aop} \leftarrow \text{EXTZ}((\text{VRA})_{i:i+7}) \\
\quad \text{bop} \leftarrow \text{EXTZ}((\text{VRB})_{i:i+7}) \\
\quad \text{VRT}_{i:i+7} \leftarrow \text{if } (\text{aop} > \text{bop}) \text{ then } (\text{VRA})_{i:i+7} \text{ else } (\text{VRB})_{i:i+7} \\
\end{array}
\]

For each integer value \( i \) from 0 to 15, do the following.
Unsigned-integer byte element \( i \) in VRA is compared to unsigned-integer byte element \( i \) in VRB. The larger of the two values is placed into byte element \( i \) of VRT.

Special Registers Altered:
None

**Vector Maximum Unsigned Doubleword VX-form**

\[ \text{vmaxud} \quad \text{VRT, VRA, VRB} \]

\[
\begin{array}{cccc}
4 & 6 & 11 & 16 & 21 & 194 \\
\end{array}
\]

\[
\begin{array}{c}
do \ i=0 \ to \ 1 \\
\quad \text{aop} \leftarrow \text{VR}[\text{VRA}].\text{dword}[i] \\
\quad \text{bop} \leftarrow \text{VR}[\text{VRB}].\text{dword}[i] \\
\quad \text{VR}[\text{VRT}].\text{dword}[i] \leftarrow \text{if } (\text{aop} > \text{bop}) \text{ then } \text{aop} \text{ else } \text{bop} \\
\end{array}
\]

For each integer value \( i \) from 0 to 1, do the following.
The unsigned integer value in doubleword element \( i \) of VR[VRT] is compared to the unsigned integer value in doubleword element \( i \) of VR[VRB]. The larger of the two values is placed into doubleword element \( i \) of VR[VRT].

Special Registers Altered:
None
Vector Maximum Signed Halfword VX-form

vmaxsh VRT,VRA,VRB

\[
\begin{array}{cccccc}
\text{do } i=0 \text{ to } 127 \text{ by } 16 \\
\text{aop } & \leftarrow & \text{EXTS}(\text{VRA})_{i:i+15} \\
\text{bop } & \leftarrow & \text{EXTS}(\text{VRB})_{i:i+15} \\
\text{VRT}_{i:i+15} & \leftarrow & \text{if } \text{aop} >_{\text{si}} \text{bop} \text{ then } \text{(VRA)}_{i:i+15} \text{ else } \text{(VRB)}_{i:i+15} \\
\end{array}
\]

For each integer value \(i\) from 0 to 7, do the following.
Signed-integer halfword element \(i\) in VRA is compared to signed-integer halfword element \(i\) in VRB. The larger of the two values is placed into halfword element \(i\) of VRT.

Special Registers Altered:
None

Vector Maximum Signed Word VX-form

vmaxsw VRT,VRA,VRB

\[
\begin{array}{cccccc}
\text{do } i=0 \text{ to } 127 \text{ by } 32 \\
\text{aop } & \leftarrow & \text{EXTS}(\text{VRA})_{i:i+31} \\
\text{bop } & \leftarrow & \text{EXTS}(\text{VRB})_{i:i+31} \\
\text{VRT}_{i:i+31} & \leftarrow & \text{if } \text{aop} >_{\text{si}} \text{bop} \text{ then } \text{(VRA)}_{i:i+31} \text{ else } \text{(VRB)}_{i:i+31} \\
\end{array}
\]

For each integer value \(i\) from 0 to 3, do the following.
Signed-integer word element \(i\) in VRA is compared to signed-integer word element \(i\) in VRB. The larger of the two values is placed into word element \(i\) of VRT.

Special Registers Altered:
None

Vector Maximum Unsigned Halfword VX-form

vmaxuh VRT,VRA,VRB

\[
\begin{array}{cccccc}
\text{do } i=0 \text{ to } 127 \text{ by } 16 \\
\text{aop } & \leftarrow & \text{EXTZ}(\text{VRA})_{i:i+15} \\
\text{bop } & \leftarrow & \text{EXTZ}(\text{VRB})_{i:i+15} \\
\text{VRT}_{i:i+15} & \leftarrow & \text{if } \text{aop} >_{\text{ui}} \text{bop} \text{ then } \text{(VRA)}_{i:i+15} \text{ else } \text{(VRB)}_{i:i+15} \\
\end{array}
\]

For each integer value \(i\) from 0 to 7, do the following.
Unsigned-integer halfword element \(i\) in VRA is compared to unsigned-integer halfword element \(i\) in VRB. The larger of the two values is placed into halfword element \(i\) of VRT.

Special Registers Altered:
None

Vector Maximum Unsigned Word VX-form

vmaxuw VRT,VRA,VRB

\[
\begin{array}{cccccc}
\text{do } i=0 \text{ to } 127 \text{ by } 32 \\
\text{aop } & \leftarrow & \text{EXTZ}(\text{VRA})_{i:i+31} \\
\text{bop } & \leftarrow & \text{EXTZ}(\text{VRB})_{i:i+31} \\
\text{VRT}_{i:i+31} & \leftarrow & \text{if } \text{aop} >_{\text{ui}} \text{bop} \text{ then } \text{(VRA)}_{i:i+31} \text{ else } \text{(VRB)}_{i:i+31} \\
\end{array}
\]

For each integer value \(i\) from 0 to 3, do the following.
Unsigned-integer word element \(i\) in VRA is compared to unsigned-integer word element \(i\) in VRB. The larger of the two values is placed into word element \(i\) of VRT.

Special Registers Altered:
None
Vector Minimum Signed Byte VX-form

\[ \text{vminsb VRT,VRA,VRB} \]

\[
\begin{array}{cccc}
4 & 6 & 11 & 16 \\
0 & 1 & 0 & 21 \\
770 & 31 & \\
\end{array}
\]

\[
\text{do } i=0 \text{ to } 127 \text{ by } 8 \]
\[
\text{aop } \leftarrow \text{EXTS(VRA)}_{i:i+7} \\
\text{bop } \leftarrow \text{EXTS(} \text{VRB)}_{i:i+7} \\
\text{VRT}_{i:i+7} \leftarrow (\text{aop} <_{ui} \text{bop}) \ ? \ (\text{VRA})_{i:i+7} : (\text{VRB})_{i:i+7} \\
\end{array}
\]

For each integer value \( i \) from 0 to 15, do the following.
Signed-integer byte element \( i \) in VRA is compared to signed-integer byte element \( i \) in VRB. The smaller of the two values is placed into byte element \( i \) of VRT.

Special Registers Altered:
None

Vector Minimum Signed Doubleword VX-form

\[ \text{vminsd VRT,VRA,VRB} \]

\[
\begin{array}{cccc}
4 & 6 & 11 & 16 \\
0 & 1 & 0 & 21 \\
962 & 31 & \\
\end{array}
\]

\[
\text{do } i = 0 \text{ to } 1 \\
\text{aop } \leftarrow \text{VR[VRA].dword[i]} \\
\text{bop } \leftarrow \text{VR[VRB].dword[i]} \\
\text{VR[VRT].dword[i]} \leftarrow (\text{EXTS(aop} <_{ui} \text{EXTS(bop))} ? \text{aop} : \text{bop} \\
\end{array}
\]

For each integer value \( i \) from 0 to 1, do the following.
The signed integer value in doubleword element \( i \) of \( \text{VR[VRA]} \) is compared to the signed integer value in doubleword element \( i \) of \( \text{VR[VRB]} \). The smaller of the two values is placed into doubleword element \( i \) of \( \text{VR[VRT]} \).

Special Registers Altered:
None

Vector Minimum Unsigned Byte VX-form

\[ \text{vminub VRT,VRA,VRB} \]

\[
\begin{array}{cccc}
4 & 6 & 11 & 16 \\
0 & 1 & 0 & 21 \\
770 & 31 & \\
\end{array}
\]

\[
\text{do } i=0 \text{ to } 127 \text{ by } 8 \]
\[
\text{aop } \leftarrow \text{EXTZ(VRA)}_{i:i+7} \\
\text{bop } \leftarrow \text{EXTZ(} \text{VRB)}_{i:i+7} \\
\text{VRT}_{i:i+7} \leftarrow (\text{aop} <_{ui} \text{bop}) \ ? \ (\text{VRA})_{i:i+7} : (\text{VRB})_{i:i+7} \\
\end{array}
\]

For each integer value \( i \) from 0 to 15, do the following.
Unsigned-integer byte element \( i \) in VRA is compared to unsigned-integer byte element \( i \) in VRB. The smaller of the two values is placed into byte element \( i \) of VRT.

Special Registers Altered:
None

Vector Minimum Unsigned Doubleword VX-form

\[ \text{vminud VRT,VRA,VRB} \]

\[
\begin{array}{cccc}
4 & 6 & 11 & 16 \\
0 & 1 & 0 & 21 \\
706 & 31 & \\
\end{array}
\]

\[
\text{do } i = 0 \text{ to } 1 \\
\text{aop } \leftarrow \text{VR[VRA].dword[i]} \\
\text{bop } \leftarrow \text{VR[VRB].dword[i]} \\
\text{VR[VRT].dword[i]} \leftarrow (\text{aop} <_{ui} \text{bop}) ? \text{aop} : \text{bop} \\
\end{array}
\]

For each integer value \( i \) from 0 to 1, do the following.
The unsigned integer value in doubleword element \( i \) of \( \text{VR[VRA]} \) is compared to the unsigned integer value in doubleword element \( i \) of \( \text{VR[VRB]} \). The smaller of the two values is placed into doubleword element \( i \) of \( \text{VR[VRT]} \).

Special Registers Altered:
None
Vector Minimum Signed Halfword VX-form

\[ \text{vminsh} \quad \text{VRT}, \text{VRA}, \text{VRB} \]

\[
\begin{array}{cccccc}
\text{do} & i=0 & \text{to} & 127 & \text{by} & 16 \\
\text{aop} & \leftarrow & \text{EXTS}(\text{VRA}_{1:15}) \\
bop & \leftarrow & \text{EXTS}(\text{VRB}_{1:15}) \\
\text{VRT}_{1:15} & \leftarrow & \text{if} & \text{aop} < \text{bop} & \text{then} & \text{VRA}_{1:15} & \text{else} & \text{VRB}_{1:15}
\end{array}
\]

For each integer value \( i \) from 0 to 7, do the following.
Signed-integer halfword element \( i \) in VRA is compared to signed-integer halfword element \( i \) in VRB. The smaller of the two values is placed into halfword element \( i \) of VRT.

Special Registers Altered:
None

Vector Minimum Signed Word VX-form

\[ \text{vminsw} \quad \text{VRT}, \text{VRA}, \text{VRB} \]

\[
\begin{array}{cccccc}
\text{do} & i=0 & \text{to} & 127 & \text{by} & 32 \\
\text{aop} & \leftarrow & \text{EXTS}(\text{VRA}_{1:31}) \\
bop & \leftarrow & \text{EXTS}(\text{VRB}_{1:31}) \\
\text{VRT}_{1:31} & \leftarrow & \text{if} & \text{aop} < \text{bop} & \text{then} & \text{VRA}_{1:31} & \text{else} & \text{VRB}_{1:31}
\end{array}
\]

For each integer value \( i \) from 0 to 3, do the following.
Signed-integer word element \( i \) in VRA is compared to signed-integer word element \( i \) in VRB. The smaller of the two values is placed into word element \( i \) of VRT.

Special Registers Altered:
None

Vector Minimum Unsigned Halfword VX-form

\[ \text{vminuh} \quad \text{VRTX}, \text{VRA}, \text{VRB} \]

\[
\begin{array}{cccccc}
\text{do} & i=0 & \text{to} & 127 & \text{by} & 16 \\
\text{aop} & \leftarrow & \text{EXTZ}(\text{VRA}_{1:15}) \\
bop & \leftarrow & \text{EXTZ}(\text{VRB}_{1:15}) \\
\text{VRTX}_{1:15} & \leftarrow & \text{if} & \text{aop} < \text{bop} & \text{then} & \text{VRA}_{1:15} & \text{else} & \text{VRB}_{1:15}
\end{array}
\]

For each integer value \( i \) from 0 to 7, do the following.
Unsigned-integer halfword element \( i \) in VRA is compared to unsigned-integer halfword element \( i \) in VRB. The smaller of the two values is placed into halfword element \( i \) of VRT.

Special Registers Altered:
None

Vector Minimum Unsigned Word VX-form

\[ \text{vminuw} \quad \text{VRTX}, \text{VRA}, \text{VRB} \]

\[
\begin{array}{cccccc}
\text{do} & i=0 & \text{to} & 127 & \text{by} & 32 \\
\text{aop} & \leftarrow & \text{EXTZ}(\text{VRA}_{1:31}) \\
bop & \leftarrow & \text{EXTZ}(\text{VRB}_{1:31}) \\
\text{VRTX}_{1:31} & \leftarrow & \text{if} & \text{aop} < \text{bop} & \text{then} & \text{VRA}_{1:31} & \text{else} & \text{VRB}_{1:31}
\end{array}
\]

For each integer value \( i \) from 0 to 3, do the following.
Unsigned-integer word element \( i \) in VRA is compared to unsigned-integer word element \( i \) in VRB. The smaller of the two values is placed into word element \( i \) of VRT.

Special Registers Altered:
None
6.9.3 Vector Integer Compare Instructions

The 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 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.

**Bit Description**
- 0: The relation is true for all element pairs (i.e., VRT is set to all 1s)
- 1: 0
- 2: The relation is false for all element pairs (i.e., VRT is set to all 0s)
- 3: 0

**Programming Note**
- `vcmpequb`, `vcmpequh`, `vcmpequw`, and `vcmpequd` can be used for unsigned or signed-integers.

### Vector Compare Equal Unsigned Byte VC-form

```plaintext
vcmpequb VRT,VRA,VRB (Rc=0)
vcmpequb. VRT,VRA,VRB (Rc=1)
```

For each integer value `i` from 0 to 15, do the following.

Unsigned-integer byte element `i` in VRA is compared to unsigned-integer byte element `i` in VRB. Byte element `i` in VRT is set to all 1s if unsigned-integer byte element `i` in VRA is equal to unsigned-integer byte element `i` in VRB, and is set to all 0s otherwise.

**Special Registers Altered:**
- CR field 6: ............................... (if Rc=1)

### Vector Compare Equal Unsigned Halfword VC-form

```plaintext
vcmpequh VRT,VRA,VRB (Rc=0)
vcmpequh. VRT,VRA,VRB (Rc=1)
```

For each integer value `i` from 0 to 7, do the following.

Unsigned-integer halfword element `i` in VRA is compared to unsigned-integer halfword element `i` in VRB. Halfword element `i` in VRT is set to all 1s if unsigned-integer halfword element `i` in VRA is equal to unsigned-integer halfword element `i` in VRB, and is set to all 0s otherwise.

**Special Registers Altered:**
- CR field 6: ............................... (if Rc=1)
Vector Compare Equal Unsigned Word
VC-form

<table>
<thead>
<tr>
<th></th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>31</td>
</tr>
</tbody>
</table>

do i=0 to 127 by 32
   \begin{align*}
   \text{VRT}[i:31] & \leftarrow (\text{VRA}[i:31] \text{ != int (VRB)[i:31]} \ ? 128 \ : 129) \\
   \text{end}
   \end{align*}

if Rc=1 then do
   \begin{align*}
   \text{t} & \leftarrow (\text{VRT}=128) \\
   \text{f} & \leftarrow (\text{VRT}=129) \\
   \text{CR}6 & \leftarrow \text{t || 0b0 || f || 0b0}
   \end{align*}

end

For each integer value \( i \) from 0 to 127, do the following.

The unsigned integer value in word element \( i \) in \( \text{VR}[\text{VRA}] \) is compared to the unsigned integer value in word element \( i \) in \( \text{VR}[\text{VRB}] \). Word element \( i \) in \( \text{VR}[\text{VRT}] \) is set to all 1s if unsigned-integer word element \( i \) in \( \text{VR}[\text{VRA}] \) is equal to unsigned-integer word element \( i \) in \( \text{VR}[\text{VRB}] \), and is set to all 0s otherwise.

Special Registers Altered:
\( \text{CR field 6} \). . . . . . . . . . . . . . . . . . . . . . . . . . . (if Rc = 1)

Vector Compare Equal Unsigned Doubleword VX-form

<table>
<thead>
<tr>
<th></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>
</tbody>
</table>

do i = 0 to 1
   \begin{align*}
   \text{aop} & \leftarrow \text{XORZ}(\text{VR}[\text{VRA}].\text{dword}[i]) \\
   \text{bop} & \leftarrow \text{XORZ}(\text{VR}[\text{VRB}].\text{dword}[i]) \\
   \text{if} \ (\text{aop} = \text{bop}) \ \text{then do}
   \begin{align*}
   \text{VR}[\text{VRT}].\text{dword}[i] & \leftarrow \text{0xFFFF_FFFF_FFFF_FFFF} \\
   \text{flag.bit}[i] & \leftarrow \text{0b1}
   \end{align*}
   \text{end}
   \text{else do}
   \begin{align*}
   \text{VR}[\text{VRT}].\text{dword}[i] & \leftarrow \text{0x0000_0000_0000_0000} \\
   \text{flag.bit}[i] & \leftarrow \text{0b0}
   \end{align*}
   \text{end}
   \end{align*}

end

if Rc=1 then do
   \begin{align*}
   \text{CR.bit}[24] & \leftarrow (\text{flag}=\text{0b11}) \\
   \text{CR.bit}[25] & \leftarrow \text{0b0} \\
   \text{CR.bit}[26] & \leftarrow (\text{flag}=\text{0b00}) \\
   \text{CR.bit}[27] & \leftarrow \text{0b0}
   \end{align*}

end

For each integer value \( i \) from 0 to 1, do the following.

The unsigned integer value in doubleword element \( i \) of \( \text{VR}[\text{VRA}] \) is compared to the unsigned integer value in doubleword element \( i \) of \( \text{VR}[\text{VRB}] \). Doubleword element \( i \) of \( \text{VR}[\text{VRT}] \) is set to all 1s if the unsigned integer value in doubleword element \( i \) of \( \text{VR}[\text{VRA}] \) is equal to the unsigned integer value in doubleword element \( i \) of \( \text{VR}[\text{VRB}] \), and is set to all 0s otherwise.

Special Registers Altered:
\( \text{CR field 6} \). . . . . . . . . . . . . . . . . . . . . . . . . . . (if Rc = 1)
Vector Compare Greater Than Signed
Byte VC-form

\[
\text{vcmpltb} \quad \text{VRT}, \text{VRA}, \text{VRB} \quad (Rc=0)
\]

\[
\text{vcmpltb} \quad \text{VRT}, \text{VRA}, \text{VRB} \quad (Rc=1)
\]

<table>
<thead>
<tr>
<th></th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>0</th>
<th>16</th>
<th>21</th>
<th>22</th>
<th>774</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>16</td>
<td>16</td>
<td>21</td>
<td>22</td>
<td>31</td>
</tr>
</tbody>
</table>

\[
\text{do } \text{i=0 to 127 by 8}\]
\[
\text{VRT}_{\text{i:i+7}} \leftarrow ((\text{VRA})_{\text{i:i+7}} >_{\text{si}} (\text{VRB})_{\text{i:i+7}}) \quad ? \ 81 \ : \ 80
\]
\text{end}

if \( \text{Rc}=1 \) then do
\[
\text{t} \leftarrow (\text{VR}_{\text{T}})^{\text{1281}}
\]
\[
\text{f} \leftarrow (\text{VR}_{\text{R}})^{\text{1280}}
\]
\[
\text{CR} \leftarrow t || f || f || f
\]
\text{end}

For each integer value \( \text{i} \) from 0 to 15, do the following. The signed integer value in byte element \( \text{i} \) in \( \text{VR}[\text{VRA}] \) is compared to the signed integer value in byte element \( \text{i} \) in \( \text{VR}[\text{VRB}] \). Byte element \( \text{i} \) in \( \text{VR}[\text{VRT}] \) is set to all 1s if signed-integer byte element \( \text{i} \) in \( \text{VR}[\text{VRA}] \) is greater than to signed-integer byte element \( \text{i} \) in \( \text{VR}[\text{VRB}] \), and is set to all 0s otherwise.

Special Registers Altered:
\[
\text{CR} \quad \text{field} \ 6 \ .......................... \ (\text{if } \text{Rc}=1)
\]

Vector Compare Greater Than Signed
Doubleword VX-form

\[
\text{vcmpltd} \quad \text{VRT}, \text{VRA}, \text{VRB} \quad (Rc=0)
\]

\[
\text{vcmpltd} \quad \text{VRT}, \text{VRA}, \text{VRB} \quad (Rc=1)
\]

<table>
<thead>
<tr>
<th></th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>0</th>
<th>16</th>
<th>21</th>
<th>22</th>
<th>967</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>16</td>
<td>16</td>
<td>21</td>
<td>22</td>
<td>31</td>
</tr>
</tbody>
</table>

\[
\text{do } \text{i=0 to 1}\]
\[
\text{aop} \leftarrow \text{EXTS}(\text{VR}[\text{VRA}].\text{dword}[\i])
\]
\[
\text{bop} \leftarrow \text{EXTS}(\text{VR}[\text{VRB}].\text{dword}[\i])
\]
if \( \text{aop} >_{\text{si}} \text{bop} \) then do
\[\text{VR}[\text{VRT}].\text{dword}[\i] \leftarrow 0xFFFF_FFFF_FFFF_FFFF\]
\[
\text{flag.bit}[\i] \leftarrow 0b1
\]
\text{end}
else do
\[\text{VR}[\text{VRT}].\text{dword}[\i] \leftarrow 0x0000_0000_0000_0000\]
\[
\text{flag.bit}[\i] \leftarrow 0b0
\]
\text{end}
end

if "vcmpltd." then do
\[
\text{CR.bit}[24] \leftarrow (\text{flag}=0b11)
\]
\[
\text{CR.bit}[25] \leftarrow 0b0
\]
\[
\text{CR.bit}[26] \leftarrow (\text{flag}=0b00)
\]
\[
\text{CR.bit}[27] \leftarrow 0b0
\]
\text{end}

For each integer value \( \text{i} \) from 0 to 1, do the following. The signed integer value in doubleword element \( \text{i} \) of \( \text{VR}[\text{VRA}] \) is compared to the signed integer value in doubleword element \( \text{i} \) of \( \text{VR}[\text{VRB}] \). Doubleword element \( \text{i} \) of \( \text{VR}[\text{VRT}] \) is set to all 1s if the signed integer value in doubleword element \( \text{i} \) of \( \text{VR}[\text{VRA}] \) is greater than the signed integer value in doubleword element \( \text{i} \) of \( \text{VR}[\text{VRB}] \), and is set to all 0s otherwise.

Special Registers Altered:
\[
\text{CR} \quad \text{field} \ 6 \ .......................... \ (\text{if } \text{Rc}=1)
Vector Compare Greater Than Signed Halfword VC-form

```
vcmpgtsh  VRT, VRA,VRB  (Rc=0)
vcmpgtsh. VRT, VRA,VRB  (Rc=1)
```

```
<table>
<thead>
<tr>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>31</td>
</tr>
<tr>
<td>16</td>
<td>21</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td></td>
<td></td>
</tr>
<tr>
<td>838</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
do i=0 to 127 by 16
end
if Rc=1 then do
   t ← (VRT=1281)
f ← (VRT=1280)
CR6 ← t || 0b0 || f || 0b0
end
```

For each integer value i from 0 to 7, do the following.

Signed-integer halfword element i in VRA is compared to signed-integer halfword element i in VRB. Halfword element i in VRT is set to all 1s if signed-integer halfword element i in VRA is greater than signed-integer halfword element i in VRB, and is set to all 0s otherwise.

Special Registers Altered:

<table>
<thead>
<tr>
<th>CR field</th>
<th>(if Rc=1)</th>
</tr>
</thead>
</table>

Vector Compare Greater Than Signed Word VC-form

```
vcmpgtsw  VRT, VRA,VRB  (Rc=0)
vcmpgtsw. VRT, VRA,VRB  (Rc=1)
```

```
<table>
<thead>
<tr>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>31</td>
</tr>
<tr>
<td>16</td>
<td>21</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td></td>
<td></td>
</tr>
<tr>
<td>902</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
do i=0 to 127 by 32
   VRT[i:i+31] ← (VRA[i:i+31] > (VRB[i:i+31]) ? 321 : 320
end
if Rc=1 then do
   t ← (VRT=321)
f ← (VRT=320)
CR6 ← t || 0b0 || f || 0b0
end
```

For each integer value i from 0 to 3, do the following.

Signed-integer word element i in VRA is compared to signed-integer word element i in VRB. Word element i in VRT is set to all 1s if signed-integer word element i in VRA is greater than signed-integer word element i in VRB, and is set to all 0s otherwise.

Special Registers Altered:

| CR field | (if Rc=1) |
Vector Compare Greater Than Unsigned Byte VC-form

\[ \text{vcmpgtub} \quad \text{VRT}, \text{VRA}, \text{VRB} \quad (\text{Rc}=0) \]
\[ \text{vcmpgtub.} \quad \text{VRT}, \text{VRA}, \text{VRB} \quad (\text{Rc}=1) \]

\[
\begin{array}{cccccccc}
4 & 6 & 11 & 16 & 21 & 22 & 31 & 518 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{array}
\]

do i=0 to 127 by 8
\[
\text{VRT}_i:i+7 \leftarrow ((\text{VRA})_{i+i+7} >_ui (\text{VRB})_{i+i+7}) \quad ? \quad 81 : 80
\]
end

if Rc=1 then do
\[
t \leftarrow (\text{VRT} >_{128} 0)
\]
\[
f \leftarrow (\text{VRT} <_{128} 0)
\]
\[
\text{CR6} \leftarrow t || 0b0 || f || 0b0
\]
end

For each integer value \( i \) from 0 to 15, do the following. Unsigned-integer byte element \( i \) in VRA is compared to unsigned-integer byte element \( i \) in VRB. Byte element \( i \) in VRT is set to all 1s if unsigned-integer byte element \( i \) in VRA is greater than to unsigned-integer byte element \( i \) in VRB, and is set to all 0s otherwise.

Special Registers Altered:
\[ \text{CR field} 6 \quad \text{.........................} \quad (\text{if Rc}=1) \]

Vector Compare Greater Than Unsigned Doubleword VX-form

\[ \text{vcmpgtud} \quad \text{VRT}, \text{VRA}, \text{VRB} \quad (\text{Rc}=0) \]
\[ \text{vcmpgtud.} \quad \text{VRT}, \text{VRA}, \text{VRB} \quad (\text{Rc}=1) \]

\[
\begin{array}{cccccccc}
4 & 6 & 11 & 16 & 21 & 22 & 31 & 711 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{array}
\]

do i = 0 to 1
\[
aop \leftarrow \text{EXTZ}(\text{VR}[\text{VRA}].\text{dword}[i])
bop \leftarrow \text{EXTZ}(\text{VR}[\text{VRB}].\text{dword}[i])
\]
if (EXTZ(aop) > EXTZ(bop)) then do
\[
\text{VR}[\text{VRT}].\text{dword}[i] \leftarrow \text{0xFFFF_FFFF_FFFF_FFFF}
\]
flag.bit[i] \( \leftarrow 0b1 \)
end
else do
\[
\text{VR}[\text{VRT}].\text{dword}[i] \leftarrow \text{0x0000_0000_0000_0000}
\]
flag.bit[i] \( \leftarrow 0b1 \)
end

end if "vcmpgtud." then do
\[
\text{CR} . \text{bit}[24] \leftarrow (\text{flag}=0b11)
\text{CR} . \text{bit}[25] \leftarrow 0b0
\text{CR} . \text{bit}[26] \leftarrow (\text{flag}=0b00)
\text{CR} . \text{bit}[27] \leftarrow 0b0
\]
end

For each integer value \( i \) from 0 to 1, do the following. The unsigned integer value in doubleword element \( i \) of VR[VRA] is compared to the unsigned integer value in doubleword element \( i \) of VR[VRB]. Doubleword element \( i \) of VR[VRT] is set to all 1s if the unsigned integer value in doubleword element \( i \) of VR[VRA] is greater than the unsigned integer value in doubleword element \( i \) of VR[VRB], and is set to all 0s otherwise.

Special Registers Altered:
\[ \text{CR field} 6 \quad (\text{if Rc}=1) \]
Vector Compare Greater Than Unsigned Halfword VC-form

```
vcmpgtuh VRT, VRA, VRB (Rc=0)
vcmpgtuh. VRT, VRA, VRB (Rc=1)
```

<table>
<thead>
<tr>
<th>4</th>
<th>6</th>
<th>31</th>
<th>96</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
<tr>
<td>22</td>
<td>31</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```
do i=0 to 127 by 16
  VRT_{i:i+15} \leftarrow (VRA_{i:i+15} >_{ui} (VRB_{i:i+15}) \ ? \ 1_{16} : 0_{16})
end
```  

if Rc=1 then do
  t \leftarrow (VRT_{:128})
  f \leftarrow (VRT_{:128})
  CR6 \leftarrow t \ || \ 0b0 \ || \ f \ || \ 0b0
end

For each integer value i from 0 to 7, do the following.
Unsigned-integer halfword element i in VRA is compared to unsigned-integer halfword element i in VRB. Halfword element i in VRT is set to all 1s if unsigned-integer halfword element i in VRA is greater than to unsigned-integer halfword element i in VRB, and is set to all 0s otherwise.

Special Registers Altered:

CR field 6 . . . . . . . . . . . . . . . . . . . . . . . . . (if Rc=1)

Vector Compare Greater Than Unsigned Word VC-form

```
vcmpgtuw VRT, VRA, VRB (Rc=0)
vcmpgtuw. VRT, VRA, VRB (Rc=1)
```

<table>
<thead>
<tr>
<th>4</th>
<th>6</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td>31</td>
</tr>
<tr>
<td>128</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```
do i=0 to 127 by 32
  VRT_{i:i+31} \leftarrow (VRA_{i:i+31} >_{ui} (VRB_{i:i+31}) \ ? \ 1_{32} : 0_{32})
end
```  

if Rc=1 then do
  t \leftarrow (VRT_{:128})
  f \leftarrow (VRT_{:128})
  CR6 \leftarrow t \ || \ 0b0 \ || \ f \ || \ 0b0
end

For each integer value i from 0 to 3, do the following.
Unsigned-integer word element i in VRA is compared to unsigned-integer word element i in VRB. Word element i in VRT is set to all 1s if unsigned-integer word element i in VRA is greater than to unsigned-integer word element i in VRB, and is set to all 0s otherwise.

Special Registers Altered:

CR field 6 . . . . . . . . . . . . . . . . . . . . . . . . . (if Rc=1)
Vector Compare Not Equal Byte VX-form

vcmpneb     VRT, VRA, VRB  (if Rc=0)
vcmpneb.    VRT, VRA, VRB  (if Rc=1)

For each integer value \( i \) from 0 to 15, do the following.

The integer value in byte element \( i \) in VR[VRA] is compared to the integer value in byte element \( i \) in VR[VRB]. The contents of byte element \( i \) in VR[VRT] are set to 0xFF if integer value in byte element \( i \) in VR[VRA] is not equal to the integer value in byte element \( i \) in VR[VRB], and are set to 0x00 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 or Zero Byte VX-form

vcmpnezb    VRT, VRA, VRB  (if Rc=0)
vcmpnezb.   VRT, VRA, VRB  (if Rc=1)

For each integer value \( i \) from 0 to 15, do the following.

The integer value in byte element \( i \) in VR[VRA] is compared to the integer value in byte element \( i \) in VR[VRB]. The contents of byte element \( i \) in VR[VRT] are set to 0xFF if integer value in byte element \( i \) in VR[VRA] is not equal to the integer value in byte element \( i \) in VR[VRB] or either value is equal to 0x00, and are set to 0x00 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 VX-form

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vcmpneh</td>
<td>VRT, VRA, VRB</td>
<td>(if Rc = 0)</td>
</tr>
<tr>
<td>vcmpneph</td>
<td>VRT, VRA, VRB</td>
<td>(if Rc = 1)</td>
</tr>
</tbody>
</table>

For each integer value \( i \) from 0 to 7, do the following.

The integer value in halfword element \( i \) in \( VRT, VRA \) is compared to the integer value in halfword element \( i \) in \( VRB \). The contents of halfword element \( i \) in \( VRT \) are set to 0xFFFF if integer value in halfword element \( i \) in \( VRA \) is not equal to the integer value in halfword element \( i \) in \( VRA \), and are set to 0x0000 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 or Zero Halfword VX-form

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vcmpnezh</td>
<td>VRT, VRA, VRB</td>
<td>(if Rc = 0)</td>
</tr>
<tr>
<td>vcmpnepzh</td>
<td>VRT, VRA, VRB</td>
<td>(if Rc = 1)</td>
</tr>
</tbody>
</table>

For each integer value \( i \) from 0 to 7, do the following.

The integer value in halfword element \( i \) in \( VRT, VRA \) is compared to the integer value in halfword element \( i \) in \( VRB \). The contents of halfword element \( i \) in \( VRT \) are set to 0xFFFF if integer value in halfword element \( i \) in \( VRA \) is not equal to the integer value in halfword element \( i \) in \( VRA \) or either value is equal to 0x0000, and are set to 0x0000 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 Word VX-form

\[
vcmpnew \quad \text{VRT, VRA, VRB} \quad (\text{if } Rc = 0) \\
vcmpnew. \quad \text{VRT, VRA, VRB} \quad (\text{if } Rc = 1)
\]

For each integer value \( i \) from 0 to 3, do the following.

The integer value in word element \( i \) in VR[VRA] is compared to the integer value in word element \( i \) in VR[VRB]. The contents of word element \( i \) in VR[VRT] are set to 0xFFFF_FFFF if integer value in word element \( i \) in VR[VRA] is not equal to the integer value in word element \( i \) in VR[VRB], and are set to 0x0000_0000 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 \((\text{if } Rc = 1)\)

Vector Compare Not Equal or Zero Word VX-form

\[
vcmpnezw \quad \text{VRT, VRA, VRB} \quad (\text{if } Rc = 0) \\
vcmpnezw. \quad \text{VRT, VRA, VRB} \quad (\text{if } Rc = 1)
\]

For each integer value \( i \) from 0 to 3, do the following.

The integer value in word element \( i \) in VR[VRA] is compared to the integer value in word element \( i \) in VR[VRB]. The contents of word element \( i \) in VR[VRT] are set to 0xFFFF_FFFF or either value is equal to 0x0000_0000 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 \((\text{if } Rc = 1)\)
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 Vector Register 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 Vector 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 Vy to register Vx.

\[
\text{vmr } Vx,Vy \quad \text{(equivalent to: } \text{vor } Vx,Vy,Vy)\]

Vector Complement Register

The Vector NOR instruction can be coded in a way such that it complements the contents of one Vector Register and places the result into another Vector Register. 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.

\[
\text{vnot } Vx,Vy \quad \text{(equivalent to: } \text{vnor } Vx,Vy,Vy)\]

Vector Logical AND VX-form

\[
vand \quad VRT,VRA,VRB
\]

\[
\begin{array}{cccccc}
4 & 6 & 11 & 16 & 21 & 31 \\
0 & 1028
\end{array}
\]

\[
\text{VR}[VRT] \leftarrow \text{VR}[VRA] \& \text{VR}[VRB]
\]

The contents of \( \text{VR} \{ \text{VRA} \} \) are ANDed with the contents of \( \text{VR} \{ \text{VRB} \} \) and the result is placed into \( \text{VR} \{ \text{VRT} \} \).

Special Registers Altered:
None

Vector Logical Equivalence VX-form

\[
veqv \quad VRT,VRA,VRB
\]

\[
\begin{array}{cccccc}
4 & 6 & 11 & 16 & 21 & 31 \\
0 & 1092
\end{array}
\]

\[
\text{VR}[VRT] \leftarrow \text{VR}[VRA] \oplus \text{VR}[VRB]
\]

The contents of \( \text{VR} \{ \text{VRA} \} \) are XORed with the contents of \( \text{VR} \{ \text{VRB} \} \) and the complemented result is placed into \( \text{VR} \{ \text{VRT} \} \).

Special Registers Altered:
None

Vector Logical AND with Complement VX-form

\[
vandc \quad VRT,VRA,VRB
\]

\[
\begin{array}{cccccc}
4 & 6 & 11 & 16 & 21 & 31 \\
0 & 1092
\end{array}
\]

\[
\text{VR}[VRT] \leftarrow \text{VR}[VRA] \& \neg \text{VR}[VRB]
\]

The contents of \( \text{VR} \{ \text{VRA} \} \) are ANDed with the complement of the contents of \( \text{VR} \{ \text{VRB} \} \) and the result is placed into \( \text{VR} \{ \text{VRT} \} \).

Special Registers Altered:
None

Vector Logical NAND VX-form

\[
vnand \quad VRT,VRA,VRB
\]

\[
\begin{array}{cccccc}
4 & 6 & 11 & 16 & 21 & 31 \\
0 & 1412
\end{array}
\]

\[
\text{if MSR.VEC}=0 \; \text{then VECTOR_UNAVAILABLE}();
\]

\[
\text{VR}[VRT] \leftarrow \neg (\text{VR}[VRA] \& \text{VR}[VRB])
\]

The contents of \( \text{VR} \{ \text{VRA} \} \) are ANDed with the contents of \( \text{VR} \{ \text{VRB} \} \) and the complemented result is placed into \( \text{VR} \{ \text{VRT} \} \).

Special Registers Altered:
None
Vector Logical OR with Complement VX-form

**vorc**  
VRT, VRA, VRB

\[
VR[VRT] \leftarrow VR[VRA] \mid \neg VR[VRB]
\]

The contents of \(VR[VRA]\) are ORed with the complement of the contents of \(VR[VRB]\) and the result is placed into \(VR[VRT]\).

Special Registers Altered:  
None

Vector Logical NOR VX-form

**vnor**  
VRT, VRA, VRB

\[
VR[VRT] \leftarrow \neg (VR[VRA] \mid VR[VRB])
\]

The contents of \(VR[VRA]\) are ORed with the contents of \(VR[VRB]\) and the complemented result is placed into \(VR[VRT]\).

Special Registers Altered:  
None

Vector Logical OR VX-form

**vor**  
VRT, VRA, VRB

\[
VR[VRT] \leftarrow VR[VRA] \mid VR[VRB]
\]

The contents of \(VR[VRA]\) are ORed with the contents of \(VR[VRB]\) and the result is placed into \(VR[VRT]\).

Special Registers Altered:  
None

Vector Logical XOR VX-form

**vxor**  
VRT, VRA, VRB

\[
VR[VRT] \leftarrow VR[VRA] \oplus VR[VRB]
\]

The contents of \(VR[VRA]\) are XORed with the contents of \(VR[VRB]\) and the result is placed into \(VR[VRT]\).

Special Registers Altered:  
None
6.9.5 Vector Parity Byte Instructions

**Vector Parity Byte Word VX-form**

`vprtybw` VRT, VRB

<table>
<thead>
<tr>
<th></th>
<th>VRT</th>
<th>VRB</th>
<th>1538</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>8</td>
<td>16</td>
</tr>
</tbody>
</table>

if MSR.VEC=0 then Vector_Unavailable()

```
for i = 0 to 3
  for j = 0 to 3
    s = 0
    s = s + VRB.word[i].byte[j].bit[7]
    VRB[word[i].byte[j].bit[7] = Chop(EXTZ(s), 32)
  end
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 VRB is odd, the value 1 is placed into word element \( i \) of VRT; otherwise the value 0 is placed into word element \( i \) of VRT.

**Special Registers Altered:**

None

**Vector Parity Byte Doubleword VX-form**

`vprtybd` VRT, VRB

<table>
<thead>
<tr>
<th></th>
<th>VRT</th>
<th>VRB</th>
<th>1538</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>8</td>
<td>16</td>
</tr>
</tbody>
</table>

if MSR.VEC=0 then Vector_Unavailable()

```
for i = 0 to 1
  for j = 0 to 7
    s = 0
    s = s + VRB.dword[i].byte[j].bit[7]
    VRB.dword[i] = Chop(EXTZ(s), 64)
  end
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 VRB is odd, the value 1 is placed into doubleword element \( i \) of VRT; otherwise the value 0 is placed into doubleword element \( i \) of VRT.

**Special Registers Altered:**

None

**Vector Parity Byte Quadword VX-form**

`vprtybq` VRT, VRB

<table>
<thead>
<tr>
<th></th>
<th>VRT</th>
<th>VRB</th>
<th>1538</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>10</td>
<td>16</td>
</tr>
</tbody>
</table>

if MSR.VEC=0 then Vector_Unavailable()

```
s = 0
for j = 0 to 15
  s = s + VRB.byte[j].bit[7]
  VRB = Chop(EXTZ(s), 128)
end
```

If the sum of the least significant bit in each byte element of VRB is odd, the value 1 is placed into VRT; otherwise the value 0 is placed into VRT.

**Special Registers Altered:**

None
6.9.6 Vector Integer Rotate and Shift Instructions

**Vector Rotate Left Byte VX-form**

```plaintext
vrlb VRT,VRA,VRB

<table>
<thead>
<tr>
<th>i</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>
```

For each integer value \( i \) from 0 to 15, do the following.

Byte element \( i \) in VRA is rotated left by the number of bits specified in the low-order 3 bits of the corresponding byte element \( i \) in VRB.

The result is placed into byte element \( i \) in VRT.

Special Registers Altered:
None

**Vector Rotate Left Word VX-form**

```plaintext
vrw VRT,VRA,VRB

<table>
<thead>
<tr>
<th>i</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>
```

For each integer value \( i \) from 0 to 3, do the following.

Word element \( i \) in VRA is rotated left by the number of bits specified in the low-order 5 bits of the corresponding word element \( i \) in VRB.

The result is placed into word element \( i \) in VRT.

Special Registers Altered:
None

**Vector Rotate Left Halfword VX-form**

```plaintext
vrh VRT,VRA,VRB

<table>
<thead>
<tr>
<th>i</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>
```

For each integer value \( i \) from 0 to 7, do the following.

Halfword element \( i \) in VRA is rotated left by the number of bits specified in the low-order 4 bits of the corresponding halfword element \( i \) in VRB.

The result is placed into halfword element \( i \) in VRT.

Special Registers Altered:
None

**Vector Rotate Left Doubleword VX-form**

```plaintext
vrld VRT,VRA,VRB

<table>
<thead>
<tr>
<th>i</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>196</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>
```

For each integer value \( i \) from 0 to 1, do the following.

The contents of doubleword element \( i \) of VRA are rotated left by the number of bits specified in bits 58:63 of doubleword element \( i \) of VRB.

The result is placed into doubleword element \( i \) of VRT.

Special Registers Altered:
None
### Vector Shift Left Byte VX-form

**vslb** \(\text{VRT}, \text{VRA}, \text{VRB}\)

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>260</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>18</td>
<td>21</td>
</tr>
</tbody>
</table>

Do i=0 to 127 by 8

\[- \text{sh} \leftarrow (\text{VRB})_{1:4,1:7} \]
\[- \text{VRT}_{1:4,7} \leftarrow (\text{VRA})_{1:4,7} \ll \text{sh} \]

For each integer value i from 0 to 15, do the following.

- Byte element i in VRA is shifted left by the number of bits specified in the low-order 3 bits of byte element i in VRB.
  - Bits shifted out of bit 0 are lost.
  - Zeros are supplied to the vacated bits on the right.

The result is placed into byte element i of VRT.

**Special Registers Altered:**

None

### Vector Shift Left Halfword VX-form

**vslh** \(\text{VRT}, \text{VRA}, \text{VRB}\)

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>324</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>18</td>
<td>21</td>
</tr>
</tbody>
</table>

Do i=0 to 127 by 16

\[- \text{sh} \leftarrow (\text{VRB})_{1:4,1:15} \]
\[- \text{VRT}_{1:4,15} \leftarrow (\text{VRA})_{1:4,15} \ll \text{sh} \]

For each integer value i from 0 to 7, do the following.

- Halfword element i in VRA is shifted left by the number of bits specified in the low-order 4 bits of halfword element i in VRB.
  - Bits shifted out of bit 0 are lost.
  - Zeros are supplied to the vacated bits on the right.

The result is placed into halfword element i of VRT.

**Special Registers Altered:**

None

### Vector Shift Left Word VX-form

**vslw** \(\text{VRT}, \text{VRA}, \text{VRB}\)

<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>18</td>
<td>21</td>
</tr>
</tbody>
</table>

Do i=0 to 127 by 32

\[- \text{sh} \leftarrow (\text{VRB})_{1:4,1:31} \]
\[- \text{VRT}_{1:4,31} \leftarrow (\text{VRA})_{1:4,31} \ll \text{sh} \]

For each integer value i from 0 to 3, do the following.

- Word element i in VRA is shifted left by the number of bits specified in the low-order 5 bits of word element i in VRB.
  - Bits shifted out of bit 0 are lost.
  - Zeros are supplied to the vacated bits on the right.

The result is placed into word element i of VRT.

**Special Registers Altered:**

None

### Vector Shift Left Doubleword VX-form

**vsld** \(\text{VRT}, \text{VRA}, \text{VRB}\)

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>1476</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>18</td>
<td>21</td>
</tr>
</tbody>
</table>

Do i=0 to 1

\[- \text{sh} \leftarrow \text{VRB}.\text{dword}[i].\text{bit}[58:63] \]
\[- \text{VRT}.\text{dword}[i] \leftarrow \text{VRA}.\text{dword}[i] \ll \text{sh} \]

For each integer value i from 0 to 1, do the following.

- The contents of doubleword element i of VRA are shifted left by the number of bits specified in bits 58:63 of doubleword element i of VRB.
  - Bits shifted out of bit 0 are lost.
  - Zeros are supplied to the vacated bits on the right.

The result is placed into doubleword element i of VRT.

**Special Registers Altered:**

None
Vector Shift Right Byte VX-form

vsrb  VRT, VRA, VRB

\[
\begin{array}{|c|c|c|c|c|}
\hline
\text{i} & \text{VRT} & \text{VRA} & \text{VRB} & \text{VSB} \\
\hline
0 & 6 & 11 & 16 & 21 \\
\hline
\end{array}
\]

\[
\text{do i=0 to 127 by 8} \\
\text{sh} \leftarrow (\text{VRB})[i+5:i+7] \\
\text{VRT}[i:i+7] \leftarrow (\text{VRA})[i:i+7] >> \text{sh} \\
\text{end}
\]

For each integer value \(i\) from 0 to 15, do the following.
Byte element \(i\) in VRA is shifted right by the number of bits specified in the low-order 3 bits of byte element \(i\) in VRB. Bits shifted out of 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 VRT.

Special Registers Altered:
None

Vector Shift Right Halfword VX-form

vsrh  VRT, VRA, VRB

\[
\begin{array}{|c|c|c|c|c|}
\hline
\text{i} & \text{VRT} & \text{VRA} & \text{VRB} & \text{VSH} \\
\hline
0 & 6 & 11 & 16 & 21 \\
\hline
\end{array}
\]

\[
\text{do i=0 to 127 by 16} \\
\text{sh} \leftarrow (\text{VRB})[i+12:i+15] \\
\text{VRT}[i:i+15] \leftarrow (\text{VRA})[i:i+15] >> \text{sh} \\
\text{end}
\]

For each integer value \(i\) from 0 to 7, do the following.
Halfword element \(i\) in VRA is shifted right by the number of bits specified in the low-order 4 bits of halfword element \(i\) in VRB. Bits shifted out of 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 VRT.

Special Registers Altered:
None

Vector Shift Right Word VX-form

vsrw  VRT, VRA, VRB

\[
\begin{array}{|c|c|c|c|c|}
\hline
\text{i} & \text{VRT} & \text{VRA} & \text{VRB} & \text{VSW} \\
\hline
0 & 6 & 11 & 16 & 21 \\
\hline
\end{array}
\]

\[
\text{do i=0 to 127 by 32} \\
\text{sh} \leftarrow (\text{VRB})[i+27:i+31] \\
\text{VRT}[i:i+31] \leftarrow (\text{VRA})[i:i+31] >> \text{sh} \\
\text{end}
\]

For each integer value \(i\) from 0 to 3, do the following.
Word element \(i\) in VRA is shifted right by the number of bits specified in the low-order 5 bits of word element \(i\) in VRB. Bits shifted out of 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 VRT.

Special Registers Altered:
None

Vector Shift Right Doubleword VX-form

vsrd  VRT, VRA, VRB

\[
\begin{array}{|c|c|c|c|c|}
\hline
\text{i} & \text{VRT} & \text{VRA} & \text{VRB} & \text{VSD} \\
\hline
0 & 6 & 11 & 16 & 21 \\
\hline
\end{array}
\]

\[
\text{do i = 0 to 1} \\
\text{sh} \leftarrow \text{VR}[\text{VRB}.dword[i].bit[58:63]} \\
\text{VR}[\text{VRT}.dword[i]] \leftarrow \text{VR}[\text{VRA}.dword[i]] >> \text{sh} \\
\text{end}
\]

For each integer value \(i\) from 0 to 1, do the following.
The contents of doubleword element \(i\) of VR[VRA] are shifted right by the number of bits specified in bits 58:63 of doubleword element \(i\) of VR[VRB]. Zeros are supplied to the vacated bits on the left. The result is placed into doubleword element \(i\) of VR[VRT].

Special Registers Altered:
None
**Vector Shift Right Algebraic Byte VX-form**

```
vsrab VRT,VRA,VRB
```

```
<table>
<thead>
<tr>
<th>i</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>31</td>
</tr>
</tbody>
</table>
```

\[
\text{do } i = 0 \text{ to } 127 \text{ by } 8 \\
sh \leftarrow (\text{VRB})_{i+5:i+7} \\
VRT_{i+5:i+7} \leftarrow (\text{VRA})_{i+5} >>_8 \text{ sh} \\
\text{end}
\]

For each integer value \(i\) from 0 to 15, do the following.

Byte element \(i\) in VRA is shifted right by the number of bits specified in the low-order 3 bits of the corresponding byte element \(i\) in VRB. Bits shifted out of bit 7 of the byte element are lost. Bit 0 of the byte element is replicated to fill the vacated bits on the left. The result is placed into byte element \(i\) of VRT.

**Special Registers Altered:**
None

**Vector Shift Right Algebraic Halfword VX-form**

```
vsrah VRT,VRA,VRB
```

```
<table>
<thead>
<tr>
<th>i</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>31</td>
</tr>
</tbody>
</table>
```

\[
\text{do } i = 0 \text{ to } 127 \text{ by } 16 \\
sh \leftarrow (\text{VRB})_{i+12:i+15} \\
VRT_{i+12:i+15} \leftarrow (\text{VRA})_{i+15} >>_4 \text{ sh} \\
\text{end}
\]

For each integer value \(i\) from 0 to 7, do the following.

Halfword element \(i\) in VRA is shifted right by the number of bits specified in the low-order 4 bits of the corresponding halfword element \(i\) in VRB. Bits shifted out of bit 15 of the halfword are lost. Bit 0 of the halfword is replicated to fill the vacated bits on the left. The result is placed into halfword element \(i\) of VRT.

**Special Registers Altered:**
None

**Vector Shift Right Algebraic Word VX-form**

```
vsraw VRT,VRA,VRB
```

```
<table>
<thead>
<tr>
<th>i</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>900</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>8</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>
```

\[
\text{do } i = 0 \text{ to } 127 \text{ by } 32 \\
sh \leftarrow (\text{VRB})_{i+27:i+31} \\
VRT_{i+27:i+31} \leftarrow (\text{VRA})_{i+31} >>_4 \text{ sh} \\
\text{end}
\]

For each integer value \(i\) from 0 to 3, do the following.

Word element \(i\) in VRA is shifted right by the number of bits specified in the low-order 5 bits of the corresponding word element \(i\) in VRB. Bits shifted out of bit 31 of the word are lost. Bit 0 of the word is replicated to fill the vacated bits on the left. The result is placed into word element \(i\) of VRT.

**Special Registers Altered:**
None

**Vector Shift Right Algebraic Doubleword VX-form**

```
vsrad VRT,VRA,VRB
```

```
<table>
<thead>
<tr>
<th>i</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>964</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>8</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>
```

\[
\text{do } i = 0 \text{ to } 1 \\
sh \leftarrow \text{VR[VRB].dword[i].bit[58:63]} \\
VRT.dword[i] \leftarrow \text{VR[VRT].dword[i]} >>_4 \text{ sh} \\
\text{end}
\]

For each integer value \(i\) from 0 to 1, do the following.

The contents of doubleword element \(i\) of VR[VRA] are shifted right by the number of bits specified in bits 58:63 of doubleword element \(i\) of VR[VRB]. Bits shifted out of bit 31 of the doubleword are lost. Bit 0 of doubleword element \(i\) of VRT[VRA] is replicated to fill the vacated bits on the left.

The result is placed into doubleword element \(i\) of VR[VRT].

**Special Registers Altered:**
None
Vector Rotate Left Word then AND with Mask VX-form

For each integer value \( i \) from 0 to 3, do the following.

Let \( \text{src1} \) be the contents of word element \( i \) of \( VR[VRA] \).

Let \( \text{src2} \) be the contents of word element \( i \) of \( VR[VRB] \).

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 \( VR[VRT] \).

Special Registers Altered:

None

Vector Rotate Left Word then Mask Insert VX-form

For each integer value \( i \) from 0 to 3, do the following.

Let \( \text{src1} \) be the contents of word element \( i \) of \( VR[VRA] \).

Let \( \text{src2} \) be the contents of word element \( i \) of \( VR[VRB] \).

Let \( \text{src3} \) be the contents of word element \( i \) of \( VR[VRT] \).

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 inserted into \( \text{src3} \) under control of the generated mask.

The result is placed into word element \( i \) of \( VR[VRT] \).

Special Registers Altered:

None
Vector Rotate Left Doubleword then AND with Mask VX-form

vrldnm  VRT,VRA,VRB

if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 1
       src1.dword[0] ← VR[VRA].dword[i]
       src1.dword[1] ← VR[VRA].dword[i]
       src2 ← VR[VRB].dword[i]
       b ← src2.bit[42:47]
       e ← src2.bit[50:55]
       n ← src2.bit[58:63]
       m ← MASK(b, e)
       VR[VRT].dword[i] ← r & n
end

For each integer value i from 0 to 1, do the following.
Let src1 be the contents of doubleword element i of VR[VRA].
Let src2 be the contents of doubleword element i of VR[VRB].
Let mb be the contents of bits 42:47 of src2.
Let me be the contents of bits 50:55 of src2.
Let sh be the contents of bits 58:63 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 doubleword element i of VR[VRT].

Special Registers Altered:
None

Vector Rotate Left Doubleword then Mask Insert VX-form

vrldmi  VRT,VRA,VRB

if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 1
       src1.dword[0] ← VR[VRA].dword[i]
       src1.dword[1] ← VR[VRA].dword[i]
       src2 ← VR[VRB].dword[i]
       src3 ← VR[VRT].dword[i]
       b ← src2.bit[42:47]
       e ← src2.bit[50:55]
       n ← src2.bit[58:63]
       m ← MASK(b, e)
       VR[VRT].dword[i] ← (r & m) | (src3 & ¬m)
end

For each integer value i from 0 to 1, do the following.
Let src1 be the contents of doubleword element i of VR[VRA].
Let src2 be the contents of doubleword element i of VR[VRB].
Let src3 be the contents of doubleword element i of VR[VRT].

Let mb be the contents of bits 42:47 of src2.
Let me be the contents of bits 50:55 of src2.
Let sh be the contents of bits 58:63 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 inserted into src3 under control of the generated mask.

The result is placed into doubleword element i of VR[VRT].

Special Registers Altered:
None
6.10 Vector Floating-Point Instruction Set

6.10.1 Vector Floating-Point Arithmetic Instructions

**Vector Add Floating-Point VX-form**

\[
vaddfp \text{ VRT}, \text{ VRA}, \text{ VRB}
\]

<table>
<thead>
<tr>
<th>4</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>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>

\[
\text{do } i=0 \text{ to } 127 \text{ by } 12
\]

\[
\text{VRT}_{i:i+31} \leftarrow \text{RoundToNearestSP}(\text{VRT}_{i:i+31} + \text{fp}(\text{VRA}_{i:i+31}))
\]

For each integer value \(i\) from 0 to 3, do the following.

- Single-precision floating-point element \(i\) in VRA is added to single-precision floating-point element \(i\) in VRB. The intermediate result is rounded to the nearest single-precision floating-point number and placed into word element \(i\) of VRT.

**Special Registers Altered:**

- None

**Vector Subtract Floating-Point VX-form**

\[
vsubfp \text{ VRT}, \text{ VRA}, \text{ VRB}
\]

<table>
<thead>
<tr>
<th>4</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>74</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>74</td>
</tr>
</tbody>
</table>

\[
\text{do } i=0 \text{ to } 127 \text{ by } 12
\]

\[
\text{VRT}_{i:i+31} \leftarrow \text{RoundToNearestSP}(\text{VRT}_{i:i+31} - \text{fp}(\text{VRB}_{i:i+31}))
\]

For each integer value \(i\) from 0 to 3, do the following.

- Single-precision floating-point element \(i\) in VRB is subtracted from single-precision floating-point element \(i\) in VRA. The intermediate result is rounded to the nearest single-precision floating-point number and placed into word element \(i\) of VRT.

**Special Registers Altered:**

- None
**Vector Multiply-Add Floating-Point VA-form**

\[ \text{vmaddfp} \quad \text{VRT, VRA, VRC, VRB} \]

<table>
<thead>
<tr>
<th>( i )</th>
<th>( VRT )</th>
<th>( VRA )</th>
<th>( VRB )</th>
<th>( VRC )</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

For each integer value \( i \) from 0 to 127, do the following.

Single-precision floating-point element \( i \) in VRA is multiplied by single-precision floating-point element \( i \) in VRC. Single-precision floating-point element \( i \) in VRB 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 VRT.

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 0/floating-point +0.0 that may already be available.

---

**Vector Negative Multiply-Subtract Floating-Point VA-form**

\[ \text{vnmsubfp} \quad \text{VRT, VRA, VRC, VRB} \]

<table>
<thead>
<tr>
<th>( i )</th>
<th>( VRT )</th>
<th>( VRA )</th>
<th>( VRB )</th>
<th>( VRC )</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

For each integer value \( i \) from 0 to 127, do the following.

Single-precision floating-point element \( i \) in VRA is multiplied by single-precision floating-point element \( i \) in VRC. Single-precision floating-point element \( i \) in VRB 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 VRT.

Special Registers Altered:

None
6.10.2 Vector Floating-Point Maximum and Minimum Instructions

**Vector Maximum Floating-Point VX-form**

```
 vmaxfp VRT,VRA,VRB
```

<table>
<thead>
<tr>
<th></th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>1034</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>11</td>
<td>16</td>
</tr>
</tbody>
</table>

```
do i=0 to 127 by 32
  gt_flag ← ( (VRA)i:i+31 > (VRB)i:i+31 )
  VRTi:i+31 ← gt_flag ? (VRA)i:i+31 : (VRB)i:i+31
end
```

For each integer value i from 0 to 3, do the following.
Single-precision floating-point element i in VRA is compared to single-precision floating-point element i in VRB. The larger of the two values is placed into word element i of VRT.

The maximum of +0 and -0 is +0. The maximum of any value and a NaN is a QNaN.

**Special Registers Altered:**
None

**Vector Minimum Floating-Point VX-form**

```
 vminfp VRT,VRA,VRB
```

<table>
<thead>
<tr>
<th></th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>1098</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

```
do i=0 to 127 by 32
  lt_flag ← ( (VRA)i:i+31 < (VRB)i:i+31 )
  VRTi:i+31 ← lt_flag ? (VRA)i:i+31 : (VRB)i:i+31
end
```

For each integer value i from 0 to 3, do the following.
Single-precision floating-point element i in VRA is compared to single-precision floating-point element i in VRB. The smaller of the two values is placed into word element i of VRT.

The minimum of +0 and -0 is -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

**Vector Convert with truncate Floating-Point To Signed Word format Saturate VX-form**

vctxs VRT,VRB,UIM

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>UIM</th>
<th>VRB</th>
<th>970</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>31</td>
</tr>
</tbody>
</table>

**do i=0 to 127 by 32**

\[
VRT_{i:i+31} \leftarrow \text{ConvertSPtoSXWsaturate}([VRB]_{i:i+31}, \ UIM)
\]

**end**

For each integer value i from 0 to 3, do the following. Single-precision floating-point word element i in VRB 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 VRT.

**Special Registers Altered:**

SAT

**Extended Mnemonics:**

Example of an extended mnemonics for Vector Convert to Signed Fixed-Point Word Saturate:

**Equivalent to:**

\[\text{vcfpsxws VRT,VRB,UIM} \quad \text{vctxs VRT,VRB,UIM}\]

**Vector Convert with truncate Floating-Point To Unsigned Word format Saturate VX-form**

vctux VRT,VRB,UIM

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>UIM</th>
<th>VRB</th>
<th>906</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>8</td>
<td>11</td>
<td>16</td>
<td>31</td>
</tr>
</tbody>
</table>

**do i=0 to 127 by 32**

\[
VRT_{i:i+31} \leftarrow \text{ConvertSPtoUXWsaturate}([VRB]_{i:i+31}, \ UIM)
\]

**end**

For each integer value i from 0 to 3, do the following. Single-precision floating-point word element i in VRB 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 VRT.

**Special Registers Altered:**

SAT

**Extended Mnemonics:**

Example of an extended mnemonics for Vector Convert to Unsigned Fixed-Point Word Saturate:

**Equivalent to:**

\[\text{vcfpuxws VRT,VRB,UIM} \quad \text{vctuxs VRT,VRB,UIM}\]
**Vector Convert with round to nearest Signed Word format VX-form**

\[ \text{vcfsx VRT,VRB,UIM} \]

<table>
<thead>
<tr>
<th></th>
<th>VRT</th>
<th>UIM</th>
<th>VRB</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
<tr>
<td>842</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

do i=0 to 127 by 32
\[ \text{VRT}_{i:i+31} \leftarrow \text{ConvertSXXtoSP}( (\text{VRB})_{i:i+31} ) \div 2^{\text{UIM}} \]
end

For each integer value \( i \) from 0 to 3, do the following.
Signed fixed-point word element \( i \) in \( \text{VRB} \) is converted to the nearest single-precision floating-point value. Each result is divided by \( 2^{\text{UIM}} \) and placed into word element \( i \) of \( \text{VRT} \).

**Special Registers Altered:**
None

**Extended Mnemonics:**
Examples of extended mnemonics for Vector Convert from Signed Fixed-Point Word

\[ \text{vcsxwfp} \quad \text{VRT,VRB,UIM} \quad \text{vcfsx} \quad \text{VRT,VRB,UIM} \]

**Vector Convert with round to nearest Unsigned Word format VX-form**

\[ \text{vcfux VRT,VRB,UIM} \]

<table>
<thead>
<tr>
<th></th>
<th>VRT</th>
<th>UIM</th>
<th>VRB</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
<tr>
<td>778</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

do i=0 to 127 by 32
\[ \text{VRT}_{i:i+31} \leftarrow \text{ConvertUXXtoSP}( (\text{VRB})_{i:i+31} ) \div 2^{\text{UIM}} \]
end

For each integer value \( i \) from 0 to 3, do the following.
Unsigned fixed-point word element \( i \) in \( \text{VRB} \) is converted to the nearest single-precision floating-point value. The result is divided by \( 2^{\text{UIM}} \) and placed into word element \( i \) of \( \text{VRT} \).

**Special Registers Altered:**
None

**Extended Mnemonics:**
Examples of extended mnemonics for Vector Convert from Unsigned Fixed-Point Word

\[ \text{vcuxwfp} \quad \text{VRT,VRB,UIM} \quad \text{vcfux} \quad \text{VRT,VRB,UIM} \]
### Vector Round to Floating-Point Integer toward -Infinity VX-form

**vrfin**  
**VRT,VRB**

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>4</th>
<th>VRT</th>
<th>VRB</th>
<th>714</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>i=0 to 127 by 32</td>
<td>VRT&lt;sub&gt;0:31&lt;/sub&gt; ← RoundToSPIntFloor( VRB&lt;sub&gt;0:31&lt;/sub&gt; )</td>
<td>end</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For each integer value i from 0 to 3, do the following.  
Single-precision floating-point element i in VRB 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 VRT.

**Special Registers Altered:**  
None

---

### Vector Round to Floating-Point Integer Nearest VX-form

**vrfin**  
**VRT,VRB**

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>4</th>
<th>VRT</th>
<th>VRB</th>
<th>522</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>i=0 to 127 by 32</td>
<td>VRT&lt;sub&gt;0:31&lt;/sub&gt; ← RoundToSPIntNear( VRB&lt;sub&gt;0:31&lt;/sub&gt; )</td>
<td>end</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For each integer value i from 0 to 3, do the following.  
Single-precision floating-point element i in VRB is rounded to a single-precision floating-point integer using the rounding mode Round to Nearest.  

The result is placed into the corresponding word element i of VRT.

**Special Registers Altered:**  
None

---

### Vector Round to Floating-Point Integer toward +Infinity VX-form

**vrfip**  
**VRT,VRB**

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>4</th>
<th>VRT</th>
<th>VRB</th>
<th>650</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>i=0 to 127 by 32</td>
<td>VRT&lt;sub&gt;0:31&lt;/sub&gt; ← RoundToSPIntCeil( VRB&lt;sub&gt;0:31&lt;/sub&gt; )</td>
<td>end</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For each integer value i from 0 to 3, do the following.  
Single-precision floating-point element i in VRB 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 VRT.

**Special Registers Altered:**  
None

---

**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 Zero VX-form

\[ \text{vr} \text{fiz} \quad \text{VRT}, \text{VRB} \]

<table>
<thead>
<tr>
<th></th>
<th>VRT</th>
<th></th>
<th>VRB</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>6</td>
<td>11</td>
<td>586</td>
<td>31</td>
</tr>
</tbody>
</table>

\[
\text{do } i=0 \text{ to } 127 \text{ by } 12 \\
\quad \text{VRT}(0:31) \leftarrow \text{RoundTo} \text{SPIntTrunc}((\text{VRB})0:31) \\
\text{end}
\]

For each integer value \( i \) from 0 to 3, do the following.

Single-precision floating-point element \( i \) in \( \text{VRB} \) is rounded to a single-precision floating-point integer using the rounding mode \text{Round toward Zero}.

The result is placed into the corresponding word element \( i \) of \( \text{VRT} \).

**Special Registers Altered:**

None
6.10.4 Vector Floating-Point Compare Instructions

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; see Section 6.9.3.

The Vector Compare Bounds Floating-Point instruction sets the target Vector Register, and 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 \(-y \leq x \leq y\).

Vector Compare Bounds Floating-Point
VC-form

\[
\begin{align*}
\text{vcmpbfp} & \quad \text{VRT,VRA,VRB} \quad (\text{Rc}=0) \\
\text{vcmpbfp} & \quad \text{VRT,VRA,VRB} \quad (\text{Rc}=1)
\end{align*}
\]

<table>
<thead>
<tr>
<th></th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>6</th>
<th>966</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>6</td>
<td>11</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td>21</td>
<td>22</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td>23</td>
<td>31</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
do \ i = 0 \ to 127 \ by \ 32 \\
& \text{le} \leftarrow (\text{VRT}_{i:i+31} \leq \text{VRB}_{i:i+31}) \\
& \text{ge} \leftarrow (\text{VRT}_{i:i+31} \geq \text{VRB}_{i:i+31}) \\
& \text{VRT}_{i:i+31} \leftarrow \text{le} \\| \text{ge} \| 0000
\end{align*}
\]

If Rc=1 then do
\[
\begin{align*}
& \text{ib} \leftarrow (\text{VRT}_{128}) \\
& \text{CR6} \leftarrow 0b00 \\| \text{ib} \\| 0b0
\end{align*}
\]

For each integer value i from 0 to 3, do the following.

Single-precision floating-point word element i in VRA is compared to single-precision floating-point word element i in VRB. A 2-bit value is formed that indicates whether the element in VRA is within the bounds specified by the element in VRB, as follows.

- Bit 0 of the 2-bit value is set to 0 if the element in VRA is less than or equal to the element in VRB, and is set to 1 otherwise.
- Bit 1 of the 2-bit value is set to 0 if the element in VRA is greater than or equal to the negation of the element in VRB, and is set to 1 otherwise.

The 2-bit value is placed into the high-order two bits of word element i of VRT and the remaining bits of element i are set to 0.

If Rc=1, CR field 6 is set as follows.

<table>
<thead>
<tr>
<th>Bit Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

Bit Description

- 2, Set to indicate whether all four elements in VRA are within the bounds specified by the corresponding element in VRB, otherwise set to 0.
- 3, Set to 0

Special Registers Altered:

- CR field 6 
  (if Rc=1)

Programming Note

Each single-precision floating-point word element in VRB should be non-negative; if it is negative, the corresponding element in VRA will necessarily be out of bounds.

One exception to this is when the value of an element in VRB is -0.0 and the value of the corresponding element in VRA is either +0.0 or -0.0. +0.0 and -0.0 compare equal to -0.0.
Vector Compare Equal Floating-Point VC-form

\[ \text{vcmp} \equiv \text{fp (VRT, VRA, VRB)} \quad (\text{Rc}=0) \]
\[ \text{vcmpf} \equiv \text{fp (VRT, VRA, VRB)} \quad (\text{Rc}=1) \]

\[
\begin{array}{cccccc}
4 & 6 & 11 & 16 & 21 & 22 & 31 \\
0 & 1 & 2 & 3 & 4 & 5 & 6 \\
\end{array}
\]

For each integer value \( i \) from 0 to 3, do the following.

Single-precision floating-point element \( i \) in VRA is compared to single-precision floating-point element \( i \) in VRB. Word element \( i \) in VRT is set to all 1s if single-precision floating-point element \( i \) in VRA is equal to single-precision floating-point element \( i \) in VRB, and is set to all 0s otherwise.

If the source element \( i \) in VRA or the source element \( i \) in VRB is a NaN, VRT is set to all 0s, indicating "not equal to". If the source element \( i \) in VRA and the source element \( i \) in VRB are both infinity with the same sign, VRT is set to all 1s, indicating "equal to".

Special Registers Altered:
\[ \text{CR field 6} \quad (\text{if Rc}=1) \]

Vector Compare Greater Than or Equal Floating-Point VC-form

\[ \text{vcmp} \equiv \text{fp (VRT, VRA, VRB)} \quad (\text{Rc}=0) \]
\[ \text{vcmpf} \equiv \text{fp (VRT, VRA, VRB)} \quad (\text{Rc}=1) \]

\[
\begin{array}{cccccc}
4 & 6 & 11 & 16 & 21 & 22 & 31 \\
0 & 1 & 2 & 3 & 4 & 5 & 6 \\
\end{array}
\]

For each integer value \( i \) from 0 to 3, do the following.

Single-precision floating-point element \( i \) in VRA is compared to single-precision floating-point element \( i \) in VRB. Word element \( i \) in VRT is set to all 1s if single-precision floating-point element \( i \) in VRA is greater than or equal to single-precision floating-point element \( i \) in VRB, and is set to all 0s otherwise.

If the source element \( i \) in VRA or the source element \( i \) in VRB is a NaN, VRT is set to all 0s, indicating "not greater than or equal to". If the source element \( i \) in VRA and the source element \( i \) in VRB are both infinity with the same sign, VRT is set to all 1s, indicating "greater than or equal to".

Special Registers Altered:
\[ \text{CR field 6} \quad (\text{if Rc}=1) \]
Vector Compare Greater Than Floating-Point VC-form

vcmpgtfp VRT,VRA,VRB (Rc=0)
vcmpgtfp. VRT,VRA,VRB (Rc=1)

\[
\begin{array}{cccccccc}
\text{do } i=0 \text{ to } 127 \text{ by } 32 \\
VRT_{i:i+31} \leftarrow (\text{VRA}_{i:i+31} >fp \text{ VRB}_{i:i+31}) \ ? \ 321 : 320 \\
\text{end} \\
\text{if } Rc=1 \text{ then do} \\
\quad t \leftarrow [\text{VRT}=1281] \\
\quad f \leftarrow [\text{VRT}=1280] \\
\quad CR6 \leftarrow t \ || \ 0b0 \ || \ f \ || \ 0b0 \\
\text{end}
\end{array}
\]

For each integer value \( i \) from 0 to 3, do the following.
Single-precision floating-point element \( i \) in VRA is compared to single-precision floating-point element \( i \) in VRB. Word element \( i \) in VRT is set to all 1s if single-precision floating-point element \( i \) in VRA is greater than single-precision floating-point element \( i \) in VRB, and is set to all 0s otherwise.

If the source element \( i \) in VRA or the source element \( i \) in VRB is a NaN, VRT is set to all 0s, indicating “not greater than”. If the source element \( i \) in VRA and the source element \( i \) in VRB are both infinity with the same sign, VRT is set to all 0s, indicating “not greater than”.

Special Registers Altered:

CR field 6 \( \ldots \ldots \ldots \ldots \ldots \quad \text{if } Rc=1 \)
6.10.5 Vector Floating-Point Estimate Instructions

**Vector 2 Raised to the Exponent Estimate Floating-Point VX-form**

\[ \text{vexptefp} \quad \text{VRT}, \text{VRB} \]

\[
\begin{array}{cccccc}
4 & VRT & / / & & & 394 \\
5 & 6 & 11 & 16 & 21 & 31 \\
\end{array}
\]

\[
\text{do } i=0 \text{ to } 127 \text{ by } 32 \\
\quad \text{VRT}_{i:i+31} \leftarrow \text{Power2EstimateSP}(\text{VRB}_{i:i+31}) \\
\text{end}
\]

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 \( i \) in \( \text{VRB} \) is placed into word element \( i \) of \( \text{VRT} \).

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

**Vector Log Base 2 Estimate Floating-Point VX-form**

\[ \text{vlogefp} \quad \text{VRT}, \text{VRB} \]

\[
\begin{array}{cccccc}
4 & VRT & / / & & & 458 \\
0 & 6 & 11 & 16 & 21 & 31 \\
\end{array}
\]

\[
\text{do } i=0 \text{ to } 127 \text{ by } 32 \\
\quad \text{VRT}_{i:i+31} \leftarrow \text{LogBase2EstimateSP}(\text{VRB}_{i:i+31}) \\
\text{end}
\]

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 \( i \) in \( \text{VRB} \) is placed into the corresponding word element of \( \text{VRT} \).

Let \( x \) be any single-precision floating-point input value. Unless \( |x-1| \leq 0.125 \) 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 register \( \text{VRT} \) 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
Vector Reciprocal Estimate
Floating-Point VX-form

vrefp VRT,VRB

\[
\text{do } i=0 \text{ to } 127 \text{ by } 32 \\
\text{VRT}_{i:i+31} \leftarrow \text{ReciprocalEstimateSP}(\text{VRB}_{i:i+31})
\]
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 \( i \) in \( \text{VRB} \) is placed into word element \( i \) of \( \text{VRT} \).

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>+0</td>
</tr>
<tr>
<td>+Infinity</td>
<td>QNaN</td>
</tr>
<tr>
<td>NaN</td>
<td>QNaN</td>
</tr>
</tbody>
</table>

Special Registers Altered:
None

Vector Reciprocal Square Root Estimate
Floating-Point VX-form

vrsqrtefp VRT,VRB

\[
\text{do } i=0 \text{ to } 127 \text{ by } 32 \\
\text{VRT}_{i:i+31} \leftarrow \text{RecipSquareRootEstimateSP}(\text{VRB}_{i:i+31})
\]
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 \( i \) in \( \text{VRB} \) is placed into word element \( i \) of \( \text{VRT} \).

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**

\[
\begin{align*}
\text{vcipher} & \quad \text{VRT, VRA, VRB} \\
4 & \quad 0 & \quad 1288 \\
11 & \quad 16 & \quad 21 & \quad 31 \\
\end{align*}
\]

\[
\begin{align*}
\text{State} & \leftarrow \text{VR}[\text{VRA}] \\
\text{RoundKey} & \leftarrow \text{VR}[\text{VRB}] \\
\text{vtemp}_1 & \leftarrow \text{SubBytes}(\text{State}) \\
\text{vtemp}_2 & \leftarrow \text{ShiftRows}(\text{vtemp}_1) \\
\text{vtemp}_3 & \leftarrow \text{MixColumns}(\text{vtemp}_2) \\
\text{VR}[\text{VRT}] & \leftarrow \text{vtemp}_3 \, ^{\oplus} \, \text{RoundKey} \\
\end{align*}
\]

Let \(\text{State}\) be the contents of \(\text{VR}[\text{VRA}]\), representing the intermediate state array during AES cipher operation.

Let \(\text{RoundKey}\) be the contents of \(\text{VR}[\text{VRB}]\), representing the round key.

One round of an AES cipher operation is performed on the intermediate \(\text{State}\) array, sequentially applying the transforms, \(\text{SubBytes}\), \(\text{ShiftRows}\), \(\text{MixColumns}\), and \(\text{AddRoundKey}\), as defined in FIPS-197.

The result is placed into \(\text{VR}[\text{VRT}]\), representing the new intermediate state of the cipher operation.

**Special Registers Altered:**

None

**Vector AES Cipher Last VX-form**

\[
\begin{align*}
\text{vcipherlast} & \quad \text{VRT, VRA, VRB} \\
4 & \quad 0 & \quad 1289 \\
11 & \quad 16 & \quad 21 & \quad 31 \\
\end{align*}
\]

\[
\begin{align*}
\text{State} & \leftarrow \text{VR}[\text{VRA}] \\
\text{RoundKey} & \leftarrow \text{VR}[\text{VRB}] \\
\text{vtemp}_1 & \leftarrow \text{SubBytes}(\text{State}) \\
\text{vtemp}_2 & \leftarrow \text{ShiftRows}(\text{vtemp}_1) \\
\text{VR}[\text{VRT}] & \leftarrow \text{vtemp}_2 \, ^{\oplus} \, \text{RoundKey} \\
\end{align*}
\]

Let \(\text{State}\) be the contents of \(\text{VR}[\text{VRA}]\), representing the intermediate state array during AES cipher operation.

Let \(\text{RoundKey}\) be the contents of \(\text{VR}[\text{VRB}]\), representing the round key.

The final round in an AES cipher operation is performed on the intermediate \(\text{State}\) array, sequentially applying the transforms, \(\text{SubBytes}\), \(\text{ShiftRows}\), \(\text{AddRoundKey}\), as defined in FIPS-197.

The result is placed into \(\text{VR}[\text{VRT}]\), representing the final state of the cipher operation.

**Special Registers Altered:**

None
**Vector AES Inverse Cipher VX-form**

\[
\begin{array}{c|c|c|c|c}
\text{vncipher} & \text{VRT}, \text{VRA}, \text{VRB} \\
\hline
4 & \text{VRT} & 11 & 16 & 21 & 1352 \\
\end{array}
\]

Let \text{State} be the contents of \text{VR[VRA]}, representing the intermediate state array during AES inverse cipher operation.

Let \text{RoundKey} be the contents of \text{VR[VRB]}, representing the round key.

One round of an AES inverse cipher operation is performed on the intermediate State array, sequentially applying the transforms, \text{InvShiftRows()}, \text{InvSubBytes()}, \text{AddRoundKey()}, and \text{InvMixColumns()}, as defined in FIPS-197.

The result is placed into \text{VR[VRT]}, representing the new intermediate state of the inverse cipher operation.

Special Registers Altered:
None

---

**Vector AES Inverse Cipher Last VX-form**

\[
\begin{array}{c|c|c|c|c}
\text{vncipherlast} & \text{VRT}, \text{VRA}, \text{VRB} \\
\hline
4 & \text{VRT} & 11 & 16 & 21 & 1353 \\
\end{array}
\]

Let \text{State} be the contents of \text{VR[VRA]}, representing the intermediate state array during AES inverse cipher operation.

Let \text{RoundKey} be the contents of \text{VR[VRB]}, representing the round key.

The final round in an AES inverse cipher operation is performed on the intermediate \text{State} array, sequentially applying the transforms, \text{InvShiftRows()}, \text{InvSubBytes()}, and \text{AddRoundKey()}, as defined in FIPS-197.

The result is placed into \text{VR[VRT]}, representing the final state of the inverse cipher operation.

Special Registers Altered:
None

---

**Vector AES SubBytes VX-form**

\[
\begin{array}{c|c|c|c|c}
\text{vsbox} & \text{VRT}, \text{VRA} \\
\hline
4 & \text{VRT} & 11 & 16 & 21 & 1480 \\
\end{array}
\]

Let \text{State} be the contents of \text{VR[VRA]}, representing the intermediate state array during AES cipher operation.

The result of applying the transform, \text{SubBytes()}, on \text{State}, as defined in FIPS-197, is placed into \text{VR[VRT]}.

Special Registers Altered:
None
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

```plaintext
vshasigmaw VRT,VRA,ST,SIX

<table>
<thead>
<tr>
<th>0</th>
<th>6</th>
<th>VRT</th>
<th>VRA</th>
<th>ST</th>
<th>SIX</th>
<th>1730</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>8</td>
<td>11</td>
<td>16</td>
<td>17</td>
<td>21</td>
<td>107</td>
</tr>
</tbody>
</table>
```

For each integer value i from 0 to 1, do the following.

When ST=0 and bit 2×i of SIX is 0, a SHA-512 σ0 function is performed on the contents of doubleword element i of VR[VRA] and the result is placed into doubleword element i of VR[VRT].

When ST=0 and bit 2×i of SIX is 1, a SHA-512 σ1 function is performed on the contents of doubleword element i of VR[VRA] and the result is placed into doubleword element i of VR[VRT].

When ST=1 and bit 2×i of SIX is 0, a SHA-512 Σ0 function is performed on the contents of doubleword element i of VR[VRA] and the result is placed into doubleword element i of VR[VRT].

When ST=1 and bit 2×i of SIX is 1, a SHA-512 Σ1 function is performed on the contents of doubleword element i of VR[VRA] and the result is placed into doubleword element i of VR[VRT].

Bits 1 and 3 of SIX are reserved.

Special Registers Altered:
None

Vector SHA-256 Sigma Word VX-form

```plaintext
vshasigmad VRT,VRA,ST,SIX

<table>
<thead>
<tr>
<th>0</th>
<th>6</th>
<th>VRT</th>
<th>VRA</th>
<th>ST</th>
<th>SIX</th>
<th>1666</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>8</td>
<td>11</td>
<td>17</td>
<td>21</td>
<td>21</td>
<td>107</td>
</tr>
</tbody>
</table>
```

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 VR[VRA] and the result is placed into word element i of VR[VRT].

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 VR[VRA] and the result is placed into word element i of VR[VRT].

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 VR[VRA] and the result is placed into word element i of VR[VRT].

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 VR[VRA] and the result is placed into word element i of VR[VRT].

Special Registers Altered:
None
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**

\[ \text{vpmsumb} \quad \text{VRT}, \text{VRA}, \text{VRB} \]

```
if MSR.VEC=0 then Vector_Unavailable()
for i = 0 to 15
    prod[i].bit[0:14] <- 0
    srcA <- VR[VRA].byte[i]
    srcB <- VR[VRB].byte[i]
    for 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
    prod[i].bit[7:0] <- srcA.bit[7:0] ^ srcB.bit[7:0]
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 \( \text{VR}[\text{VRA}] \) and the contents of byte element \( i \) of \( \text{VR}[\text{VRB}] \).

For each integer value \( i \) from 0 to 7, do the following.

The exclusive-OR of \( \text{prod}[2i] \) and \( \text{prod}[2i+1] \) is placed in bits 1:15 of halfword element \( i \) of \( \text{VR}[\text{VRT}] \). Bit 0 of halfword element \( i \) of \( \text{VR}[\text{VRT}] \) is set to 0.

**Special Registers Altered:**
None

**Vector Polynomial Multiply-Sum Doubleword VX-form**

\[ \text{vpmsumd} \quad \text{VRT}, \text{VRA}, \text{VRB} \]

```
if MSR.VEC=0 then Vector_Unavailable()
for i = 0 to 1
    prod[i].bit[0:126] <- 0
    srcA <- VR[VRA].doubleword[i]
    srcB <- VR[VRB].doubleword[i]
    for 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
end
```

Let \( \text{prod}[0] \) be the 127-bit result of a binary polynomial multiplication of the contents of doubleword element 0 of \( \text{VR}[\text{VRA}] \) and the contents of doubleword element 0 of \( \text{VR}[\text{VRB}] \).

Let \( \text{prod}[1] \) be the 127-bit result of a binary polynomial multiplication of the contents of doubleword element 1 of \( \text{VR}[\text{VRA}] \) and the contents of doubleword element 1 of \( \text{VR}[\text{VRB}] \).

The exclusive-OR of \( \text{prod}[0] \) and \( \text{prod}[1] \) is placed in bits 1:127 of \( \text{VR}[\text{VRT}] \). Bit 0 of \( \text{VR}[\text{VRT}] \) is set to 0.

**Special Registers Altered:**
None
Vector Polynomial Multiply-Sum Halfword
VX-form

\[
v_{\text{pmsumh}}(\text{VRT}, \text{VRA}, \text{VRB})
\]

<table>
<thead>
<tr>
<th>4</th>
<th>0</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>1096</th>
</tr>
</thead>
</table>

For each integer value \(i\) from 0 to 7, do the following.

Let \(\text{prod}[i]\) be the 31-bit result of a binary polynomial multiplication of the contents of halfword element \(i\) of \(\text{VR}[\text{VRA}]\) and the contents of halfword element \(i\) of \(\text{VR}[\text{VRB}]\).

For each integer value \(i\) from 0 to 3, do the following.

The exclusive-OR of \(\text{prod}[2i]\) and \(\text{prod}[2i+1]\) is placed in bits 1:31 of word element \(i\) of \(\text{VR}[\text{VRT}]\). Bit 0 of word element \(i\) of \(\text{VR}[\text{VRT}]\) is set to 0.

Special Registers Altered:
None

Vector Polynomial Multiply-Sum Word
VX-form

\[
v_{\text{pmsumw}}(\text{VRT}, \text{VRA}, \text{VRB})
\]

<table>
<thead>
<tr>
<th>4</th>
<th>0</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>1160</th>
</tr>
</thead>
</table>

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 \(\text{VR}[\text{VRA}]\) and the contents of word element \(i\) of \(\text{VR}[\text{VRB}]\).

For each integer value \(i\) from 0 to 1, do the following.

The exclusive-OR of \(\text{prod}[2i]\) and \(\text{prod}[2i+1]\) is placed in bits 1:63 of doubleword element \(i\) of \(\text{VR}[\text{VRT}]\). Bit 0 of doubleword element \(i\) of \(\text{VR}[\text{VRT}]\) is set to 0.

Special Registers Altered:
None
6.11.4 Vector Permute and Exclusive-OR Instruction

**Vector Permute and Exclusive-OR**

**VA-form**

```
vpermxor VRT,VRA,VRB,VRC
```

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>VRC</th>
<th>45</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>26</td>
</tr>
<tr>
<td>31</td>
<td>31</td>
<td>31</td>
<td>31</td>
<td>31</td>
<td>31</td>
</tr>
</tbody>
</table>

```
do i = 0 to 15
    indexA ← VR[VRC].byte[i].bit[0:3]
    indexB ← VR[VRC].byte[i].bit[4:7]
    src1 ← VR[VRA].byte[indexA]
    src2 ← VR[VRB].byte[indexB]
    VSR[VRT].byte[i] ← src1 ^ src2
end
```

For each integer value \( i \) from 0 to 15, do the following.

Let \( \text{indexA} \) be the contents of bits 0:3 of byte element \( i \) of \( VR[VRC] \).

Let \( \text{indexB} \) be the contents of bits 4:7 of byte element \( i \) of \( VR[VRC] \).

The exclusive OR of the contents of byte element \( \text{indexA} \) of \( VR[VRA] \) and the contents of byte element \( \text{indexB} \) of \( VR[VRB] \) is placed into byte element \( i \) of \( VR[VRT] \).

**Special Registers Altered:**

None
6.12 Vector Gather Instruction

**Vector Gather Bits by Bytes by Doubleword VX-form**

vgbbd VRT,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>

```
for i = 0 to 1
  for j = 0 to 7
    for k = 0 to 7
      b ← VSR[VRB].dword[i].byte[k].bit[j]
      VSR[VRT].dword[i].byte[j].bit[k] ← b
  end
end
```

Let src be the contents of VR[VRB], 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 × 8-bit bit-matrix transpose is performed on the contents of each doubleword element of VR[VRB] (see Figure 104).

For each integer value i from 0 to 1, do the following.

The contents of bit 0 of each byte of doubleword element i of VR[VRB] are concatenated and placed into byte 1 of doubleword element i of VR[VRT].

The contents of bit 2 of each byte of doubleword element i of VR[VRB] are concatenated and placed into byte 2 of doubleword element i of VR[VRT].

The contents of bit 3 of each byte of doubleword element i of VR[VRB] are concatenated and placed into byte 3 of doubleword element i of VR[VRT].

The contents of bit 4 of each byte of doubleword element i of VR[VRB] are concatenated and placed into byte 4 of doubleword element i of VR[VRT].

The contents of bit 5 of each byte of doubleword element i of VR[VRB] are concatenated and placed into byte 5 of doubleword element i of VR[VRT].

The contents of bit 6 of each byte of doubleword element i of VR[VRB] are concatenated and placed into byte 6 of doubleword element i of VR[VRT].

The contents of bit 7 of each byte of doubleword element i of VR[VRB] are concatenated and placed into byte 7 of doubleword element i of VR[VRT].

**Special Registers Altered:**

None

Figure 104. Vector Gather Bits by Bytes by Doubleword
6.13 Vector Count Leading Zeros Instructions

**Vector Count Leading Zeros Byte VX-form**

```
vclzb VRT,VRB

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>///</th>
<th>VRB</th>
<th>1794</th>
</tr>
</thead>
</table>
```

if MSR.VEC=0 then Vector_Unavailable()

**Vector Count Leading Zeros Halfword VX-form**

```
vclzh VRT,VRB

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>///</th>
<th>VRB</th>
<th>1858</th>
</tr>
</thead>
</table>
```

if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 7

n ← 0

do while n < 8

if VR[VRB].byte[i].bit[n] = 0b1 then leave

n ← n + 1
end

VSR[VRT].byte[i] ← n

end

**Vector Count Leading Zeros Word VX-form**

```
vclzw VRT,VRB

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>///</th>
<th>VRB</th>
<th>1922</th>
</tr>
</thead>
</table>
```

if MSR.VEC=0 then Vector_Unavailable()

**Vector Count Leading Zeros Doubleword VX-form**

```
vclzd VRT,VRB

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>///</th>
<th>VRB</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) & (VR[VRB].dword[i].bit<n] = 0b0)

n ← n + 1
end

VSR[VRT].dword[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 VR[VRB] is placed into byte element $i$ of VR[VRT]. This number ranges from 0 to 8, inclusive.

**Special Registers Altered:**

None

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 VR[VRB] is placed into halfword element $i$ of VR[VRT]. This number ranges from 0 to 16, inclusive.

**Special Registers Altered:**

None

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 VR[VRB] is placed into word element $i$ of VR[VRT]. This number ranges from 0 to 32, inclusive.

**Special Registers Altered:**

None

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 VR[VRB] is placed into doubleword element $i$ of VR[VRT]. This number ranges from 0 to 64, inclusive.

**Special Registers Altered:**

None
6.14 Vector Count Trailing Zeros Instructions

**Vector Count Trailing Zeros Byte VX-form**

vctzb VRT,VRB

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>28</th>
<th>VRB</th>
<th>1538</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td></td>
<td>11</td>
<td></td>
<td>1538</td>
</tr>
</tbody>
</table>

if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 15
n ← 0
do while n < 8
if VR[VRB].byte[i].bit[7-n] = 0b1 then leave
n ← n + 1end
VR[VRT].byte[i] ← Chop(EXTZ(n), 8)
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 7 of byte element i of VR[VRB] is
placed into byte element i of VR[VRT]. This number
ranges from 0 to 8, inclusive.

**Special Registers Altered:**
None

**Vector Count Trailing Zeros Halfword VX-form**

vctzh VRT,VRB

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>29</th>
<th>VRB</th>
<th>1538</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td></td>
<td>11</td>
<td></td>
<td>1538</td>
</tr>
</tbody>
</table>

if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 7
n ← 0
do while n < 16
if VR[VRB].hword[i].bit[15-n] = 0b1 then leave
n ← n + 1end
VR[VRT].hword[i] ← Chop(EXTZ(n), 16)
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 15 of halfword element i of VR[VRB] is
placed into halfword element i of VR[VRT]. This number
ranges from 0 to 32, inclusive.

**Special Registers Altered:**
None

**Vector Count Trailing Zeros Word VX-form**

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>5</td>
<td></td>
<td>11</td>
<td></td>
<td>1538</td>
</tr>
</tbody>
</table>

if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
n ← 0
do while n < 32
if VR[VRB].word[i].bit[31-n] = 0b1 then leave
n ← n + 1end
VR[VRTX].word[i] ← Chop(EXTZ(n), 32)
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 VR[VRB] is
placed into word element i of VR[VRTX]. This number
ranges from 0 to 32, inclusive.

**Special Registers Altered:**
None

**Vector Count Trailing Zeros Doubleword VX-form**

vctzd VRT,VRB

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>31</th>
<th>VRB</th>
<th>1538</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td></td>
<td>11</td>
<td></td>
<td>1538</td>
</tr>
</tbody>
</table>

if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 1
n ← 0
do while n < 64
if VR[VRB].dword[i].bit[63-n] = 0b1 then leave
n ← n + 1end
VR[VRT].dword[i] ← Chop(EXTZ(n), 64)
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 63 of doubleword element i of VR[VRB] is
placed into doubleword element i of VR[VRTX]. This number
ranges from 0 to 64, inclusive.

**Special Registers Altered:**
None
### 6.14.1 Vector Count Leading/Trailing Zero LSB Instructions

**Vector Count Leading Zero Least-Significant Bits Byte VX-form**

\[
vclzlsbb \quad RT, VRB
\]

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>16</th>
<th>21</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>RT</td>
<td>0</td>
<td>1538</td>
</tr>
</tbody>
</table>

Let \( \text{count} \) be the number of contiguous leading byte elements in \( VR[VRB] \) having a zero least-significant bit. \( \text{count} \) is placed into \( \text{GPR}[RT] \).

**Special Registers Altered:**

None

**Vector Count Trailing Zero Least-Significant Bits Byte VX-form**

\[
vctzlsbb \quad RT, VRB
\]

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>16</th>
<th>21</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>RT</td>
<td>1</td>
<td>1538</td>
</tr>
</tbody>
</table>

if MSR.VEC=0 then Vector_Unavailable()

\( \text{count} \leftarrow 0 \)

do while \( \text{count} < 16 \)
  if \( VR[VRB].\text{byte}[\text{count}].\text{bit}[7]=1 \) break
  \( \text{count} \leftarrow \text{count} + 1 \)
end

\( \text{GPR}[RT] \leftarrow \text{EXTZ64}(\text{count}) \)

Let \( \text{count} \) be the number of contiguous trailing byte elements in \( VR[VRB] \) having a zero least-significant bit. \( \text{count} \) is placed into \( \text{GPR}[RT] \).

**Special Registers Altered:**

None
6.14.2 Vector Extract Element Instructions

**Vector Extract Unsigned Byte Left-Indexed VX-form**

vextublx  RT,RA,VRB

| 4 | RT | 6 | RA | 11 | VRB | 16 | 21 | 1549 | 31 |

if MSR.VEC=0 then Vector_Unavailable()  
index ← GPR[RA].bit[60:63]  
GPR[RT] ← EXTZ64(VR[VRB].byte[index])

Let index be the contents of bits 60:63 of GPR[RA].

The contents of byte element index of VR[VRB] 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 Left-Indexed VX-form**

vextuhlx  RT,RA,VRB

| 4 | RT | 6 | RA | 11 | VRB | 16 | 21 | 1613 | 31 |

if MSR.VEC=0 then Vector_Unavailable()  
index ← GPR[RA].bit[60:63]  
GPR[RT] ← EXTZ64(VR[VRB].byte[index:index+1])

Let index be the contents of bits 60:63 of GPR[RA].

The contents of byte elements index:index+1 of VR[VRB] 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 index is greater than 14, the results are undefined.

**Special Registers Altered:**

None

**Vector Extract Unsigned Byte Right-Indexed VX-form**

vextubrx  RT,RA,VRB

| 4 | RT | 6 | RA | 11 | VRB | 16 | 21 | 1805 | 31 |

if MSR.VEC=0 then Vector_Unavailable()  
index ← GPR[RA].bit[60:63]  
GPR[RT] ← EXTZ64(VR[VRB].byte[15-index])

Let index be the contents of bits 60:63 of GPR[RA].

The contents of byte element 15-index of VR[VRB] 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 Right-Indexed VX-form**

vextuhrx  RT,RA,VRB

| 4 | RT | 6 | RA | 11 | VRB | 16 | 21 | 1869 | 31 |

if MSR.VEC=0 then Vector_Unavailable()  
index ← GPR[RA].bit[60:63]  
GPR[RT] ← EXTZ64(VR[VRB].byte[14-index:15-index])

Let index be the contents of bits 60:63 of GPR[RA].

The contents of byte elements 14-index:15-index of VR[VRB] 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 index is greater than 14, the results are undefined.

**Special Registers Altered:**

None
Vector Extract Unsigned Word
Left-Indexed VX-form
vextuwx    RT,RA,VRB

Let index be the contents of bits 60:63 of GPR[RA].
The contents of byte elements index:index+3 of VR[VRB] are placed into bits 32:63 of GPR[RT].
The contents of bits 0:31 of GPR[RT] are set to 0.
If the value of index is greater than 12, the results are undefined.

Special Registers Altered:
  None

Vector Extract Unsigned Word
Right-Indexed VX-form
vextuwrx   RT,RA,VRB

Let index be the contents of bits 60:63 of GPR[RA].
The contents of byte elements index:index+3 of VR[VRB] are placed into bits 32:63 of GPR[RT].
The contents of bits 0:31 of GPR[RT] are set to 0.
If the value of index is greater than 12, the results are undefined.

Special Registers Altered:
  None
6.15 Vector Population Count Instructions

**Vector Population Count Byte VX-form**

vpopcntb \( VRT, VRB \)

<table>
<thead>
<tr>
<th>4</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>1795</th>
</tr>
</thead>
</table>

if MSR.VEC=0 then Vector_Unavailable()

\[
\text{do } i = 0 \text{ to } 15 \\
\quad n \leftarrow 0 \\
\quad \text{do } j = 0 \text{ to } 7 \\
\quad 
\quad n \leftarrow n + VR[VRB].\text{byte}[i].\text{bit}[j] \\
\text{end} \\
\text{VSR}[VRT].\text{byte}[i] \leftarrow n \\
\text{end}
\]

For each integer value \( i \) from 0 to 15, do the following.
A count of the number of bits set to 1 in byte element \( i \) of \( VR[VRB] \) is placed into byte element \( i \) of \( VR[VRT] \). This number ranges from 0 to 8, inclusive.

**Special Registers Altered:**

None

**Vector Population Count Halfword VX-form**

vpopcnth \( VRT, VRB \)

<table>
<thead>
<tr>
<th>4</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>1859</th>
</tr>
</thead>
</table>

if MSR.VEC=0 then Vector_Unavailable()

\[
\text{do } i = 0 \text{ to } 7 \\
\quad n \leftarrow 0 \\
\quad \text{do } j = 0 \text{ to } 63 \\
\quad 
\quad n \leftarrow n + VR[VRB].\text{dword}[i].\text{bit}[j] \\
\text{end} \\
\text{VSR}[VRT].\text{dword}[i] \leftarrow n \\
\text{end}
\]

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 \( VR[VRB] \) is placed into halfword element \( i \) of \( VR[VRT] \). This number ranges from 0 to 16, inclusive.

**Special Registers Altered:**

None

**Vector Population Count Doubleword VX-form**

vpopcntd \( VRT, VRB \)

<table>
<thead>
<tr>
<th>4</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>1987</th>
</tr>
</thead>
</table>

if MSR.VEC=0 then Vector_Unavailable()

\[
\text{do } i = 0 \text{ to } 1 \\
\quad n \leftarrow 0 \\
\quad \text{do } j = 0 \text{ to } 63 \\
\quad 
\quad n \leftarrow n + VR[VRB].\text{dword}[i].\text{bit}[j] \\
\text{end} \\
\text{VSR}[VRT].\text{dword}[i] \leftarrow n \\
\text{end}
\]

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 \( VR[VRB] \) is placed into doubleword element \( i \) of \( VR[VRT] \). This number ranges from 0 to 64, inclusive.

**Special Registers Altered:**

None

**Vector Population Count Word VX-form**

vpopcntw \( VRT, VRB \)

<table>
<thead>
<tr>
<th>4</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>1923</th>
</tr>
</thead>
</table>

if MSR.VEC=0 then Vector_Unavailable()

\[
\text{do } i = 0 \text{ to } 3 \\
\quad n \leftarrow 0 \\
\quad \text{do } j = 0 \text{ to } 31 \\
\quad 
\quad n \leftarrow n + VR[VRB].\text{word}[i].\text{bit}[j] \\
\text{end} \\
\text{VSR}[VRT].\text{word}[i] \leftarrow n \\
\text{end}
\]

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 \( VR[VRB] \) is placed into word element \( i \) of \( VR[VRT] \). This number ranges from 0 to 32, inclusive.

**Special Registers Altered:**

None
6.16 Vector Bit Permute Instruction

**Vector Bit Permute Doubleword VX-form**

vbpermd VRT,VRA,VRB

<table>
<thead>
<tr>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>1484</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>21</td>
</tr>
</tbody>
</table>

if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 1
    do j = 0 to 7
        index ← VR[VRB].dword[i].byte[j]
        if index < 64 then
            perm.bit[j] ← VR[VRA].dword[i].bit[index]
        else
            perm.bit[j] ← 0
        end
    end
VR[VRT].dword[i] ← EXTZ4(perm)
endo

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 VR[VRB].

If $index$ is less than 64, then the contents of bit $index$ of doubleword element $i$ of VR[VRA] are placed into bit $56+j$ of doubleword element $i$ of VR[VRT]. Otherwise, bit $56+j$ of doubleword element $i$ of VR[VRT] is set to 0.

The contents of bits 0:55 of doubleword element $i$ of VR[VRT] are set to 0.

**Special Registers Altered:** None

**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
```

**Vector Bit Permute Quadword VX-form**

vbpermq VRT,VRA,VRB

<table>
<thead>
<tr>
<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>21</td>
</tr>
</tbody>
</table>

if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 15
    do j = 0 to 7
        index ← VR[VRB].dword[i].byte[j]
        if index < 128 then
            perm.bit[j] ← VR[VRA].dword[i].bit[index]
        else
            perm.bit[j] ← 0
        end
    end
VR[VRT].dword[i] ← EXTZ64(perm)
endo

For each integer value $i$ from 0 to 15, do the following.

Let $index$ be the contents of byte element $i$ of VR[VRB].

If $index$ is less than 128, then the contents of bit $index$ of doubleword element $i$ of VR[VRA] are placed into bit $48+i$ of doubleword element $i$ of VR[VRT]. Otherwise, bit $48+i$ of doubleword element $i$ of VR[VRT] is set to 0.

The contents of bits 0:47 of VR[VRT] are set to 0.

The contents of bits 64:127 of VR[VRT] are set to 0.

**Special Registers Altered:** None

The contents of bits 0:47 of VR[VRT] are set to 0.

The contents of bits 64:127 of VR[VRT] are set to 0.
## 6.17 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 \(0b1101\).

### 6.17.1 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 Vector Register (i.e., \texttt{bcdadd}, \texttt{bcdsub}).
**Decimal Add Modulo VX-form**

bcdadd. VRT,VRA,VRB,PS

<table>
<thead>
<tr>
<th>4</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>0</td>
<td>VR</td>
<td>VRT</td>
<td>VRA</td>
<td>VRB</td>
<td>PS</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

if MSR.VEC=0 then Vector_Unavailable()

VR[VRT] ← Signed_BCD_Add(VR[VRA], VR[VRB], PS)

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] ← oz_flag | inv_flag

Let src1 be the decimal integer value in VR[VRA].
Let src2 be the decimal integer value in VR[VRB].

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.
  CR field 6 is set to 0b0010. 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.
  CR field 6 is set to 0b1001. 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 VR[VRT].

The sign code is placed in bits 124:127 of VR[VRT].

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>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>22</th>
<th>23</th>
<th>65</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>VR</td>
<td>VRT</td>
<td>VRA</td>
<td>VRB</td>
<td>PS</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

if MSR.VEC=0 then Vector_Unavailable()

VR[VRT] ← Signed_BCD_Subtract(VR[VRA], VR[VRB], PS)

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] ← oz_flag | inv_flag

Let src1 be the decimal integer value in VR[VRA].
Let src2 be the decimal integer value in VR[VRB].

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.
  CR field 6 is set to 0b0010. 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 0b0101. Otherwise, CR field 6 is set to 0b1000.

  The sign code of the result is set to 0b1101.
  If the operation overflows, CR field 6 is set to 0b0101. 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 VR[VRT].

The sign code is placed in bits 124:127 of VR[VRT].

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
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 VR. 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 VR halves of the Vector-Scalar Registers.

\texttt{gew} and \texttt{fmrgow} are provided to support direct move operations in 32-bit mode.

\begin{Verbatim}
\texttt{bcdsub. vTmp,vA,vB,0} can be used to compare decimal operands \(vA\) and \(vB\). Bits 0:2 of CR field 6 will be set to indicate \(vA\) is less than \(vB\) (LT), \(vA\) is greater than \(vB\) (GT), and \(vA\) is equal to \(vB\) (EQ).
\end{Verbatim}

\texttt{bcdsub. vTmp,vA,vA,0} can be used to test if an operand \(vA\) is an \textit{invalid encoding} of a decimal value.

\begin{Verbatim}
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}.
\end{Verbatim}
### 6.17.2 Decimal Integer Format Conversion Instructions

**Decimal Convert From National VX-form**

<table>
<thead>
<tr>
<th>bcdcfn.</th>
<th>VRT, VRB, PS</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
</tr>
</tbody>
</table>

Let `src` be the national decimal value in `VR[VRB]`.

`src` is placed in `VR[VRT]` 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 `VR[VRT]` 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 `VR[VRT]`.

For `PS=0`, the contents of nibble element 31 (i.e. sign code) of `VR[VRT]` are set to `0x0C` for positive values and to `0x0D` for negative values.

For `PS=1`, the contents of nibble element 31 (i.e. sign code) of `VR[VRT]` are set to `0x0F` for positive values and to `0x0D` 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 `VR[VRT]` are undefined and CR field 6 is set to `0b0001`.

**Special Registers Altered:**

CR field 6
Decimal Convert From Zoned VX-form

Let src be the zoned decimal value in VR[VRB].

src is placed in VR[VRT] in packed decimal format.

When 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 bits 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 0x4, 0x5, 0x6, 0x7, 0xC, 0xD, 0xE, or 0xF are interpreted as negative values.

When 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 bits 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 0xA, 0xC, 0xE, or 0xF are interpreted as positive values.

Zoned decimal source operands having a sign code of 0xB or 0xD 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 VR[VRT] are set to 0x0.

For each integer value i from 0 to 15,
- The contents of nibble 1 of byte element i of src are placed into nibble element i+15 of VR[VRT].

CR field 6 is set to reflect src compared to zero.

If src is an invalid encoding of a zoned decimal value, the contents of VR[VRT] are undefined and CR field 6 is set to 0b0001.

Special Registers Altered:
CR field 6
Decimal Convert To National VX-form

Let \( \text{src} \) be the packed decimal value in \( \text{VR[VRB]} \).

\( \text{src} \) is placed into \( \text{VR[VRT]} \) 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 \( \text{VR[VRT]} \).

The contents of nibble element \( i+24 \) of \( \text{VR[VRB]} \) are placed into nibble 3 of halfword element \( i \) of \( \text{VR[VRT]} \).

The contents of halfword element 7 (i.e., sign code) of \( \text{VR[VRT]} \) are set to \( 0x002B \) for positive values and to \( 0x002D \) for negative values.

\( \text{CR} \) field 6 is set to reflect \( \text{src} \) compared to zero, including whether or not \( \text{src} \) is too large to be represented in national decimal format.

If \( \text{src} \) is an invalid encoding of a packed decimal value, the contents of \( \text{VR[VRT]} \) are undefined and \( \text{CR} \) field 6 is set to \( 0b0001 \).

Special Registers Altered:

- \( \text{CR} \) field 6

<table>
<thead>
<tr>
<th>bcdctn.</th>
<th>VRT</th>
<th>VRB</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>5</td>
<td>16</td>
</tr>
<tr>
<td>0</td>
<td>21</td>
<td>22</td>
</tr>
<tr>
<td>385</td>
<td>31</td>
<td></td>
</tr>
</tbody>
</table>

if MSR.VEC=0 then Vector_Unavailable() 

\( \text{ox\_flag} \leftarrow 0 \)

\( \text{do } i = 0 \text{ to } 33 \)

\( \text{ox\_flag} \leftarrow \text{ox\_flag} \text{ | (VR[VRB].nibble[i] ! = 0x0)} \) 

\( \text{end} \)

\( \text{inv\_flag} \leftarrow (\text{VR[VRB].nibble[31] < 0xA}) \)

\( \text{do } i = 0 \text{ to } 30 \)

\( \text{inv\_flag} \leftarrow \text{inv\_flag} \text{ | (VR[VRB].nibble[i] > 0x9)} \)

\( \text{end} \)

\( \text{src\_sign} \leftarrow (\text{VR[VRB].nibble[31] = 0xB}) \text{ | (VR[VRB].nibble[31] = 0xD}) \)

\( \text{eq\_flag} \leftarrow (\text{VR[VRB].nibble[0:30] = 0}) \)

\( \text{lt\_flag} \leftarrow (\text{eq\_flag}=0) \text{ & (src\_sign}=1) \)

\( \text{gt\_flag} \leftarrow (\text{eq\_flag}=0) \text{ & (src\_sign}=0) \)

\( \text{do } i = 0 \text{ to } 6 \)

\( \text{result.hword}[i]\text{nibble}[0:2] \leftarrow 0x003 \)

\( \text{result.hword}[i]\text{nibble}[3] \leftarrow \text{VR[VRB].nibble}[i+24] \)

\( \text{end} \)

\( \text{result.hword}[7] \leftarrow \text{src\_sign} \text{ ? 0x002D : 0x002B} \)

\( \text{VR[VRT]} \leftarrow \text{inv\_flag} \text{ ? undefined : result} \)

\( \text{CR.bit}[56] \leftarrow \text{inv\_flag} \text{ ? 0b0 : lt\_flag} \)

\( \text{CR.bit}[57] \leftarrow \text{inv\_flag} \text{ ? 0b0 : gt\_flag} \)

\( \text{CR.bit}[58] \leftarrow \text{inv\_flag} \text{ ? 0b0 : eq\_flag} \)

\( \text{CR.bit}[59] \leftarrow \text{inv\_flag} \text{ | ox\_flag} \)
**Decimal Convert To Zoned VX-form**

\[ \text{bc dctz. VRT, VRB, PS} \]

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>4</th>
<th>VRB</th>
<th>PS</th>
<th>385</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

If MSR.VEC=0 then Vector_Unavailable().

\[ \text{inv}_{\text{flag}} \leftarrow (VR[VRB].\text{nibble}[31] < 0xA) \]

\[ \text{do } i = 0 \text{ to } 30 \]

\[ \text{inv}_{\text{flag}} \leftarrow \text{inv}_{\text{flag}} | (VR[VRB].\text{nibble}[i] > 0x9) \]

end

\[ \text{ox}_{\text{flag}} \leftarrow 0 \]

\[ \text{do } i = 0 \text{ to } 15 \]

\[ \text{ox}_{\text{flag}} \leftarrow \text{ox}_{\text{flag}} | (VR[VRB].\text{nibble}[i] \neq 0x0) \]

end

\[ \text{src}_{\text{sign}} \leftarrow (VR[VRB].\text{nibble}[31] = 0x0) | (VR[VRB].\text{nibble}[31] = 0xA) | (VR[VRB].\text{nibble}[31] = 0xC) | (VR[VRB].\text{nibble}[31] = 0xE) | (VR[VRB].\text{nibble}[31] = 0xF) \]

\[ \text{eq}_{\text{flag}} \leftarrow (VR[VRB].\text{nibble}[0:30] = 0) \]

\[ \text{lt}_{\text{flag}} \leftarrow (\text{eq}_{\text{flag}} = 0) \& (\text{src}_{\text{sign}} = 1) \]

\[ \text{gt}_{\text{flag}} \leftarrow (\text{eq}_{\text{flag}} = 0) \& (\text{src}_{\text{sign}} = 0) \]

\[ \text{do } i = 0 \text{ to } 14 \]

\[ \text{result}.\text{byte}[i].\text{nibble}[0] \leftarrow (\text{PS}=0) ? 0x3 : 0xF \]

\[ \text{result}.\text{byte}[i].\text{nibble}[1] \leftarrow VR[VRB].\text{nibble}[i+15] \]

end

if \text{src}_{\text{sign}}=0 then

\[ \text{result}.\text{byte}[15].\text{nibble}[0] \leftarrow (\text{PS}=0) ? 0x3 : 0xC \]

else

\[ \text{result}.\text{byte}[15].\text{nibble}[0] \leftarrow (\text{PS}=0) ? 0x7 : 0xD \]

end

\[ VR[VRT] \leftarrow \text{inv}_{\text{flag}} \? \text{undefined} : \text{result} \]

\[ \text{CR.bit}[56] \leftarrow \text{inv}_{\text{flag}} \? 0b0 : \text{lt}_{\text{flag}} \]

\[ \text{CR.bit}[57] \leftarrow \text{inv}_{\text{flag}} \? 0b0 : \text{gt}_{\text{flag}} \]

\[ \text{CR.bit}[58] \leftarrow \text{inv}_{\text{flag}} \? 0b0 : \text{eq}_{\text{flag}} \]

\[ \text{CR.bit}[59] \leftarrow \text{inv}_{\text{flag}} \? 0b0 : \text{eq}_{\text{flag}} \]

Let \text{src} be the packed decimal value in \text{VR[VRB]}.

\text{src} is placed into \text{VR[VRT]} in zoned 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^{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 \text{i} from 0 to 15, do the following.

The rightmost nibble of each digit \text{i} of the zoned decimal result is set to the contents of nibble \text{i+15} of \text{src}.

The result is placed into \text{VR[VRT]}.

\text{CR field 6} is set to reflect \text{src} compared to zero, including whether or not \text{src} is too large to be represented in zoned decimal format.

If \text{src} is an invalid encoding of a packed decimal value, the contents of \text{VR[VRT]} are undefined and \text{CR field 6} is set to 0b000001.

**Special Registers Altered:**

- \text{CR field 6}
Decimal Convert From Signed Quadword
VX-form

Let src be the signed integer value in VR[VRB].

src is placed into VR[VRT] in signed packed decimal format.

For PS=0, the contents of nibble element 31 (i.e., sign code) of VR[VRT] are set to 0xC for values greater than or equal to 0 and to 0xD for values less than 0.

For PS=1, the contents of nibble element 31 (i.e., sign code) of VR[VRT] 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 VR[VRB] is greater than \(10^{31.1}\) or less than \(-10^{32.1}\), the value is too large to be represented in packed decimal format, and the contents of VR[VRT] are undefined.

CR field 6 is set to reflect src compared to zero and whether or not src is too large in magnitude to be represented in packed decimal format.

Special Registers Altered:
CR field 6

Decimal Convert To Signed Quadword
VX-form

Let src be the packed decimal value in VR[VRB].

src is placed into VR[VRT] 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 VR[VRT] are undefined and CR field 6 is set to 0b00001.

Special Registers Altered:
CR field 6
Vector Multiply-by-10 Unsigned Quadword VX-form

\[ \text{vmul10uq} \quad \text{VRT, VRA} \]

1. If MSR.VEC = 0 then Vector_Unavailable.
2. \( \text{src} \leftarrow \text{EXTZ(VR[VRA])} \)
3. \( \text{prod} \leftarrow (\text{src} \ll 3) + (\text{src} \ll 1) \)
4. \( \text{VR[VRT]} \leftarrow \text{Chop(prod, 128)} \)

Let \( \text{src} \) be the unsigned integer value in VR[VRA].

The rightmost 128 bits of the product of \( \text{src} \) multiplied by the value 10 are placed into VR[VRT].

Special Registers Altered:
None

Vector Multiply-by-10 & write Carry Unsigned Quadword VX-form

\[ \text{vmul10cuq} \quad \text{VRT, VRA} \]

1. If MSR.VEC = 0 then Vector_Unavailable.
2. \( \text{src} \leftarrow \text{EXTZ(VR[VRA])} \)
3. \( \text{prod} \leftarrow (\text{src} \ll 3) + (\text{src} \ll 1) \)
4. \( \text{VR[VRT]} \leftarrow \text{Chop(prod, 128)} \)

Let \( \text{src} \) be the unsigned integer value in VR[VRA].

The product of \( \text{src} \) multiplied by the value 10 is shifted right by 128 bits. The rightmost 128 bits of the shifted result is placed into VR[VRT].

Special Registers Altered:
None

Vector Multiply-by-10 Extended Unsigned Quadword VX-form

\[ \text{vmul10euq} \quad \text{VRT, VRA, VRB} \]

1. If MSR.VEC = 0 then Vector_Unavailable.
2. \( \text{src} \leftarrow \text{EXTZ(VR[VRA])} \)
3. Let \( \text{cin} \) be the unsigned packed decimal value in bits 124:127 of VR[VRB]. Values of \( \text{cin} \) greater than 9 are undefined.
4. \( \text{prod} \leftarrow (\text{src} \ll 3) + (\text{src} \ll 1) + \text{cin} \)
5. \( \text{VR[VRT]} \leftarrow \text{Chop(prod, 128)} \)

Let \( \text{src} \) be the unsigned integer value in VR[VRA].

Let \( \text{cin} \) be the unsigned packed decimal value in bits 124:127 of VR[VRB]. Values of \( \text{cin} \) greater than 9 are undefined.

The rightmost 128 bits of the sum of \( \text{cin} \) and the product of \( \text{src} \) multiplied by the value 10 are placed into VR[VRT].

Special Registers Altered:
None

Vector Multiply-by-10 Extended & write Carry Unsigned Quadword VX-form

\[ \text{vmul10ecuq} \quad \text{VRT, VRA, VRB} \]

1. If MSR.VEC = 0 then Vector_Unavailable.
2. \( \text{src} \leftarrow \text{EXTZ(VR[VRA])} \)
3. Let \( \text{cin} \) be the unsigned packed decimal value in bits 124:127 of VR[VRB]. Values of \( \text{cin} \) greater than 9 are undefined.
4. \( \text{prod} \leftarrow (\text{src} \ll 3) + (\text{src} \ll 1) + \text{cin} \)
5. \( \text{VR[VRT]} \leftarrow \text{Chop(prod, 128)} \)

Let \( \text{src} \) be the unsigned integer value in VR[VRA].

Let \( \text{cin} \) be the unsigned packed decimal value in bits 124:127 of VR[VRB]. Values of \( \text{cin} \) greater than 9 are undefined.

The sum of \( \text{cin} \) and the product of \( \text{src} \) multiplied by the value 10 is shifted right by 128 bits. The rightmost 128 bits of the shifted result is placed into VR[VRT].

Special Registers Altered:
None
### Decimal Copy Sign VX-form

**bcdcpsgn.** VRT, VRA, VRB

<table>
<thead>
<tr>
<th>0</th>
<th>4</th>
<th>VRT</th>
<th>11</th>
<th>VRA</th>
<th>16</th>
<th>VRB</th>
<th>21</th>
<th>833</th>
<th>31</th>
</tr>
</thead>
</table>

if MSR.VEC=0 then Vector.Unavailable()

inv_flag ← (VR[VRA].nibble[31] < 0xA) | (VR[VRB].nibble[31] < 0xA)
do i = 0 to 30
   inv_flag ← inv_flag | (VR[VRA].nibble[i] > 0x9) | (VR[VRB].nibble[i] > 0x9)
end

src_sign ← (VR[VRB].nibble[31] = 0xB) | (VR[VRB].nibble[31] = 0xD)
eq_flag ← (VR[VRA].nibble[0:30] = 0)
lit_flag ← (eq_flag=0) & (src_sign=0)
gt_flag ← (eq_flag=0) & (src_sign=0)
result.nibble[0:30] ← VR[VRA].nibble[0:30]
result.nibble[31] ← VR[VRB].nibble[31]
VR[VRT] ← inv_flag ? undefined : result

CR.bit[56] ← inv_flag ? 0b0 : lit_flag
CR.bit[57] ← inv_flag ? 0b0 : gt_flag
CR.bit[58] ← inv_flag ? 0b0 : eq_flag
CR.bit[59] ← inv_flag

The decimal value in VR[VRA] is placed into VR[VRT] with the sign code of the decimal value in VR[VRB].

CR field 6 is set to reflect the result compared to zero.

If either the decimal value in VR[VRA] or the decimal value in VR[VRB] is an invalid encoding, the contents of VR[VRT] are undefined and CR field 6 is set to 0b00001.

**Special Registers Altered:**
CR field 6

### Decimal Set Sign VX-form

**bcdsetsgn.** VRT, VRB, PS

<table>
<thead>
<tr>
<th>0</th>
<th>4</th>
<th>VRT</th>
<th>11</th>
<th>VRB</th>
<th>16</th>
<th>31</th>
<th>21</th>
<th>385</th>
<th>31</th>
</tr>
</thead>
</table>

if MSR.VEC=0 then Vector.Unavailable()

inv_flag ← (VR[VRB].nibble[31] < 0xA)
do i = 0 to 30
   inv_flag ← inv_flag | (VR[VRB].nibble[i] > 0x9)
end

src_sign ← (VR[VRB].nibble[31] = 0xB) | (VR[VRB].nibble[31] = 0xD)
eq_flag ← (VR[VRB].nibble[0:30] = 0)
lit_flag ← (eq_flag=0) & (src_sign=0)
gt_flag ← (eq_flag=0) & (src_sign=0)
result.nibble[0:30] ← VR[VRB].nibble[0:30]
result.nibble[31] ← (src_sign=0) ? ((PS=0) ? 0xC:0xF) : 0xD
VR[VRT] ← inv_flag ? undefined : result

CR.bit[56] ← inv_flag ? 0b0 : lit_flag
CR.bit[57] ← inv_flag ? 0b0 : gt_flag
CR.bit[58] ← inv_flag ? 0b0 : eq_flag
CR.bit[59] ← inv_flag

Let src be the packed decimal value in VR[VRB].

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 VR[VRT] with the sign code set to 0xD.

If src is positive and PS=0, src is placed into VR[VRT] with the sign code set to 0xC.

If src is positive and PS=1, src is placed into VR[VRT] 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 VR[VRT] are undefined and CR field 6 is set to 0b00001.

**Special Registers Altered:**
CR field 6
**Decimal Shift VX-form**

<table>
<thead>
<tr>
<th>VRT, VRA, VRB, PS</th>
</tr>
</thead>
<tbody>
<tr>
<td>193   16  11  6  4</td>
</tr>
</tbody>
</table>

**bcds.**

Let \( n \) be the signed integer value in byte element 7 of VR[VRA].

Let \( \text{src} \) be the signed packed decimal value in VR[VRB].

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, \( \text{src} \) 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.

If the packed decimal value in VR[VRB] is negative, the sign code of the result is set to 0b1101.

If the packed decimal value in VR[VRB] 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 VR[VRT].

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 VR[VRT] are undefined and CR field 6 is set to 0b0001.

**Special Registers Altered:**

CR field 6
**Decimal Unsigned Shift VX-form**

<table>
<thead>
<tr>
<th>bcdus.</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>1/2/3</th>
<th>129</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>22</td>
<td>23</td>
</tr>
</tbody>
</table>

Let \( n \) be the signed integer value in byte element 7 of VR[VRA].

Let \( src \) be the unsigned packed decimal value in VR[VRB].

A valid encoding of an unsigned packed decimal value requires the contents of each nibble 0-31 must be in the range 0x0 to 0x9.

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.

The shifted result is placed into VR[VRT].

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 VR[VRT] are undefined and CR field 6 is set to 0b0000.

**Special Registers Altered:**
- CR field 6

```cpp
if MSR.VEC=0 then Vector_Unavailable();

n ← EXTs(VR[VRA].byte[7])
inv_flag ← 0
do i = 0 to 31
    inv_flag ← inv_flag | (VR[VRB].nibble[i] > 0x9)
end

eq_flag ← (VR[VRB].nibble[0:31] = 0)
gt_flag ← (eq_flag=0)
if (n > 0) then do // shift left
    shcnt ← (n<32) ? n : 32
    src.nibble[0:32] ← VR[VRB]
    src.nibble[32:63] ← DUP(0b0000,32)
    result ← src.nibble[shcnt:shcnt+32]
    ox_flag ← (shcnt > 0) & (src.nibble[0:shcnt-1] != 0)
end
else do // shift right
    shcnt ← ((¬n+1)<32) ? (¬n+1) : 32
    src.nibble[0:32] ← DUP(0b0000,32)
    src.nibble[32:63] ← VR[VRB]
    result ← src.nibble[32-shcnt:63-shcnt]
    ox_flag ← 0
end
VR[VRT] ← 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
```
Decimal Shift and Round VX-form

Let \( n \) be the signed integer value in byte element 7 of VR[VRA].

Let \( src \) be the signed packed decimal value in VR[VRB].

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 digit 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 VR[VRT].

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 VR[VRT] are undefined and CR field 6 is set to \( 0b0001 \).

Special Registers Altered:
CR field 6
### 6.17.5 Decimal Integer Truncate Instructions

#### Decimal Truncate VX-form

```
<table>
<thead>
<tr>
<th>bcdtrunc.</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>PS</th>
<th>257</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>6</td>
<td>16</td>
<td>21</td>
<td>22</td>
</tr>
</tbody>
</table>
```

Let `length` be the integer value in bits 48:63 of `VR[VRA]`.

Let `src` be the signed decimal value in `VR[VRB]`.

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 `src` is negative, the sign code of the result is set to 0b11101.

If `src` is positive, the sign code of the result is set to 0b11100 if `PS=0` and is set to 0b11111 if `PS=1`.

`src` is copied into `VR[VRT]` with the leftmost `31-length` digits each set to 0b0000. If any of the leftmost `31-length` digits of the signed decimal value in `VR[VRB]` 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 `VR[VRT]` are undefined and CR field 6 is set to 0b00001.

**Special Registers Altered:**

CR field 6
Let :math:`\text{length}` be the integer value in bits 48:63 of :math:`\text{VR}[\text{VRA}]`.

Let :math:`\text{src}` be the unsigned decimal value in :math:`\text{VR}[\text{VRB}]`.

A valid encoding of a packed decimal source operand requires the contents of each nibble 0-31 must be a value in the range :math:`0x0` to :math:`0x9`.

:math:`\text{src}` is copied into :math:`\text{VR}[\text{VRT}]` with the leftmost :math:`\text{length}` digits each set to :math:`0b0000`. If any of the leftmost :math:`\text{length}` digits of the signed decimal value in :math:`\text{VR}[\text{VRB}]` are non-zero, an overflow occurs.

CR field 6 is set to reflect :math:`\text{src}` compared to zero, including whether or not significant digits were truncated.

If :math:`\text{src}` is an invalid encoding of a packed decimal value, the contents of :math:`\text{VR}[\text{VRT}]` are undefined and CR field 6 is set to :math:`0b00001`.

**Special Registers Altered:**

- CR field 6

**Decimal Unsigned Truncate VX-form**

<table>
<thead>
<tr>
<th></th>
<th>VRB</th>
<th>VRA</th>
<th>VRT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td>11</td>
<td>6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>inv_flag</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>eq_flag</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>gt_flag</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ox_flag</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

if MSR.VEC=0 then Vector_Unavailable()"

inv_flag ← 0
do i ← 0 to 31
   inv_flag ← inv_flag | (VR[VRB].nibble[i] > 0x9)
end

length ← VR[VRA].bit[48:63]
ox_flag ← 0

eq_flag ← (VR[VRB].nibble[0:31] = 0)
gt_flag ← (VR[VRB].nibble[0:31] != 0)

if length < 32 then do
   do i ← 0 to 31-length
      if VR[VRB].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] ← VR[VRB].nibble[i]
      end
   end
end

else result ← VR[VRB]

VR[VRT] ← 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 :
6.18 Vector Status and Control Register Instructions

*Move To Vector Status and Control Register VX-form*

mtvscr     VRB

\[ \begin{array}{cccccc}
0 & 4 & 6 & 11 & 16 & 21 & 31 \\
&  &  &  &  &  & 1604 \\
\end{array} \]

\[ \text{VSCR} \leftarrow (VRB)_{96:127} \]

The contents of word element 3 of VRB are placed into the VSCR.

**Special Registers Altered:**
None

*Move From Vector Status and Control Register VX-form*

mfvscr     VRT

\[ \begin{array}{cccccc}
0 & 4 & 6 & 11 & 16 & 21 & 31 \\
&  &  &  &  &  & 1540 \\
\end{array} \]

\[ \text{VRT} \leftarrow 0 \ (\text{VSCR}) \]

The contents of the VSCR are placed into word element 3 of VRT.

The remaining word elements in VRT are set to 0.

**Special Registers Altered:**
None
Chapter 7. Vector-Scalar Floating-Point Operations

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.

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.

7.1.1.1 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 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 undefined.

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.

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.
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, halfword, word, doubleword, or quadword between storage and a VSR.

<table>
<thead>
<tr>
<th>VSR[0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>VSR[1]</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>VSR[62]</td>
</tr>
<tr>
<td>VSR[63]</td>
</tr>
</tbody>
</table>

Figure 105. Vector-Scalar Registers

<table>
<thead>
<tr>
<th>SD/UD/MD/DP 0</th>
<th>SD/UD/MD/DP 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>SW/UW/MW/SP 0</td>
<td>SW/UW/MW/SP 1</td>
</tr>
<tr>
<td>SW/UW/MW/SP 2</td>
<td>SW/UW/MW/SP 3</td>
</tr>
<tr>
<td>HP 0</td>
<td>HP 1</td>
</tr>
<tr>
<td>HP 2</td>
<td>HP 3</td>
</tr>
<tr>
<td>HP 4</td>
<td>HP 5</td>
</tr>
<tr>
<td>HP 6</td>
<td>HP 7</td>
</tr>
</tbody>
</table>

Figure 106. Vector-Scalar Register Elements

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 undefined.
<table>
<thead>
<tr>
<th>VSR[0]</th>
<th>FPR[0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>VSR[1]</td>
<td>FPR[1]</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>VSR[31]</td>
<td>FPR[31]</td>
</tr>
<tr>
<td>VSR[32]</td>
<td></td>
</tr>
<tr>
<td>VSR[33]</td>
<td></td>
</tr>
<tr>
<td>VSR[62]</td>
<td></td>
</tr>
<tr>
<td>VSR[63]</td>
<td></td>
</tr>
</tbody>
</table>

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
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 mcrfs, mtfsfi, mtfsf, or 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.

FEX 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>Bits</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:28</td>
<td>Decimal Floating-Point Rounding Control (DRN)</td>
</tr>
<tr>
<td>32</td>
<td>Floating-Point Exception Summary (FX)</td>
</tr>
<tr>
<td>33</td>
<td>Floating-Point Enabled Exception Summary (FEX)</td>
</tr>
<tr>
<td>34</td>
<td>Floating-Point Invalid Operation Exception Summary (VX)</td>
</tr>
<tr>
<td>35</td>
<td>Floating-Point Overflow Exception (OX)</td>
</tr>
<tr>
<td>36</td>
<td>Floating-Point Underflow Exception (UX)</td>
</tr>
<tr>
<td>37</td>
<td>Floating-Point Zero Divide Exception (ZX)</td>
</tr>
<tr>
<td>38</td>
<td>Floating-Point Inexact Exception (XX)</td>
</tr>
</tbody>
</table>

**Programming Note**

Access to Move To FPSCR and Move From FPSCR instructions requires FP=1.

FEX and VX are defined not to be altered implicitly by mtfsfi and mtfsf because permitting these instructions to alter FEX implicitly can cause a paradox. An example is an mtfsfi or mtfsf instruction that supplies 0 for FEX and 1 for OX, and is executed when OX=0. See also the Programming Notes with the definition of these two instructions.
<table>
<thead>
<tr>
<th>Bits</th>
<th>Definition</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>39</td>
<td>Floating-Point Invalid Operation Exception (SNAN) (VXSNAN)</td>
<td>Floating-Point Invalid Operation Exception (Inf*Zero) (VXIZE)</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 390.</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 390.</td>
</tr>
<tr>
<td></td>
<td>This bit can be set to 0 or 1 by a Move To FPSCR class instruction.</td>
<td>This bit can be set to 0 or 1 by a Move To FPSCR class instruction.</td>
</tr>
<tr>
<td>40</td>
<td>Floating-Point Invalid Operation Exception (Inf/Inf) (VXISI)</td>
<td>Floating-Point Invalid Operation Exception (Invalid Compare) (VXVC)</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 390.</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 390.</td>
</tr>
<tr>
<td></td>
<td>This bit can be set to 0 or 1 by a Move To FPSCR class instruction.</td>
<td>This bit can be set to 0 or 1 by a Move To FPSCR class instruction.</td>
</tr>
<tr>
<td>41</td>
<td>Floating-Point Invalid Operation Exception (Inf/Inf) (VXIDI)</td>
<td>Floating-Point Fraction Rounded (FR)</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 390.</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 381. This bit is not sticky.</td>
</tr>
<tr>
<td></td>
<td>This bit can be set to 0 or 1 by a Move To FPSCR class instruction.</td>
<td></td>
</tr>
<tr>
<td>42</td>
<td>Floating-Point Invalid Operation Exception (Zero÷Zero) (VXZDZ)</td>
<td>Floating-Point Fraction Inexact (FI)</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 390.</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 rounded result is inexact or the instruction caused a disabled Overflow exception. See Section 7.3.2.6 on page 381. This bit is not sticky.</td>
</tr>
<tr>
<td></td>
<td>This bit can be set to 0 or 1 by a Move To FPSCR class instruction.</td>
<td>See the definition of XX, above, regarding the relationship between FI and XX.</td>
</tr>
</tbody>
</table>

368 Power ISA™ I
### Bits Definition

<table>
<thead>
<tr>
<th>Bits</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>47:51</td>
<td><strong>Floating-Point Result Flags (FPRF)</strong></td>
</tr>
<tr>
<td></td>
<td>VSX Scalar Floating-Point Arithmetic, VSX Scalar DP-SP Conversion,</td>
</tr>
<tr>
<td></td>
<td>VSX Scalar Convert Integer to Double-Precision, and</td>
</tr>
<tr>
<td></td>
<td>VSX Scalar Round to Double-Precision Integer class instructions set</td>
</tr>
<tr>
<td></td>
<td>this field based on the result placed into the target register and</td>
</tr>
<tr>
<td></td>
<td>on the target precision, except that if any portion of the result is</td>
</tr>
<tr>
<td></td>
<td>undefined then the value placed into FPRF is undefined.</td>
</tr>
</tbody>
</table>

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>Bits</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>47</td>
<td><strong>Floating-Point Result Class Descriptor (C)</strong></td>
</tr>
<tr>
<td></td>
<td>VSX Scalar Floating-Point Arithmetic, VSX Scalar DP-SP Conversion,</td>
</tr>
<tr>
<td></td>
<td>VSX Scalar Convert Integer to Double-Precision, and</td>
</tr>
<tr>
<td></td>
<td>VSX Scalar Round to Double-Precision Integer class instructions set</td>
</tr>
<tr>
<td></td>
<td>this bit with the FPCC bits, to indicate the class of the result as</td>
</tr>
<tr>
<td></td>
<td>shown in Table 2, “Floating-Point Result Flags,” on page 371.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bits</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>48:51</td>
<td><strong>Floating-Point Condition Code (FPCC)</strong></td>
</tr>
<tr>
<td></td>
<td>VSX Scalar Compare Double-Precision instruction sets one of the FPCC</td>
</tr>
<tr>
<td></td>
<td>bits to 1 and the other three FPCC bits to 0 based on the relative</td>
</tr>
<tr>
<td></td>
<td>values of the operands being compared.</td>
</tr>
</tbody>
</table>

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 the C bit, to indicate the class of the result as shown in Table 2, “Floating-Point Result Flags,” on page 371. 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.

<table>
<thead>
<tr>
<th>Bits</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>48</td>
<td><strong>Floating-Point Less Than or Negative (FL)</strong></td>
</tr>
<tr>
<td>49</td>
<td><strong>Floating-Point Greater Than or Positive (FG)</strong></td>
</tr>
<tr>
<td>50</td>
<td><strong>Floating-Point Equal or Zero (FE)</strong></td>
</tr>
<tr>
<td>51</td>
<td><strong>Floating-Point Unordered or NaN (FU)</strong></td>
</tr>
</tbody>
</table>

### Bits Definition

<table>
<thead>
<tr>
<th>Bits</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>52</td>
<td>Reserved</td>
</tr>
<tr>
<td>53</td>
<td><strong>Floating-Point Invalid Operation Exception (Software-Defined Condition) (VXSOFT)</strong></td>
</tr>
<tr>
<td></td>
<td>This bit can be altered only by mcrfs, mtfsfi, mtfsf, mtfsb0, or</td>
</tr>
<tr>
<td></td>
<td>mtfsb1. See Section 7.4.1, “Floating-Point Invalid Operation Exception”</td>
</tr>
<tr>
<td></td>
<td>on page 390.</td>
</tr>
</tbody>
</table>

**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.

<table>
<thead>
<tr>
<th>Bits</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>54</td>
<td><strong>Floating-Point Invalid Operation Exception (Invalid Square Root) (VXSQRT)</strong></td>
</tr>
<tr>
<td></td>
<td>This bit is set to 1 when a VSX Scalar Floating-Point Arithmetic or</td>
</tr>
<tr>
<td></td>
<td>VSX Vector Floating-Point Arithmetic class instruction causes a</td>
</tr>
<tr>
<td></td>
<td>Invalid Square Root type Invalid Operation exception. See Section 7.4.1</td>
</tr>
<tr>
<td></td>
<td>, “Floating-Point Invalid Operation Exception” on page 390.</td>
</tr>
<tr>
<td></td>
<td>This bit can be set to 0 or 1 by a Move To FPSCR class instruction.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bits</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>55</td>
<td><strong>Floating-Point Invalid Operation Exception (Invalid Integer Convert) (VXCVI)</strong></td>
</tr>
<tr>
<td></td>
<td>This bit is set to 1 when a VSX Scalar Convert Double-Precision to</td>
</tr>
<tr>
<td></td>
<td>Integer, VSX Vector Convert Double-Precision to Integer, or VSX Vector</td>
</tr>
<tr>
<td></td>
<td>Convert Single-Precision to Integer class instruction causes a</td>
</tr>
<tr>
<td></td>
<td>Invalid Integer Convert type Invalid Operation exception. See Section</td>
</tr>
<tr>
<td></td>
<td>7.4.1, “Floating-Point Invalid Operation Exception” on page 390.</td>
</tr>
<tr>
<td></td>
<td>This bit can be set to 0 or 1 by a Move To FPSCR class instruction.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bits</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>56</td>
<td><strong>Floating-Point Invalid Operation Exception Enable (E)</strong></td>
</tr>
<tr>
<td></td>
<td>This bit is used by VSX Scalar Floating-Point and VSX Vector Floating-</td>
</tr>
<tr>
<td></td>
<td>Point class instructions to enable trapping on Invalid Operation</td>
</tr>
<tr>
<td></td>
<td>exceptions. See Section 7.4.1, “Floating-Point Invalid Operation</td>
</tr>
<tr>
<td></td>
<td>Exception” on page 390.</td>
</tr>
</tbody>
</table>

---

Chapter 7. Vector-Scalar Floating-Point Operations   369
### Floating-Point Overflow Exception Enable (OE)

This bit is used by VSX Scalar Floating-Point and VSX Vector Floating-Point class instructions to enable trapping on Overflow exceptions. See Section 7.4.3, “Floating-Point Overflow Exception” on page 404.

### Floating-Point Underflow Exception Enable (UE)

This bit is used by VSX Scalar Floating-Point and VSX Vector Floating-Point class instructions to enable trapping on Underflow exceptions. See Section 7.4.4, “Floating-Point Underflow Exception” on page 409.

### Floating-Point Zero Divide Exception Enable (ZE)

This bit is used by VSX Scalar Floating-Point and VSX Vector Floating-Point class instructions to enable trapping on Zero Divide exceptions. See Section 7.4.2, “Floating-Point Zero Divide Exception” on page 401.

### Floating-Point Inexact Exception Enable (XE)

This bit is used by VSX Scalar Floating-Point and VSX Vector Floating-Point class instructions to enable trapping on Inexact exceptions. See Section 7.4.5, “Floating-Point Inexact Exception” on page 414.

### Floating-Point Non-IEEE Mode (NI)

When the processor is in floating-point non-IEEE mode, the remaining FPSCR bits is permitted to 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 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 is permitted to 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 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 (RN)

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 381.

<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>
<tr>
<td>Result Flags</td>
<td>Result Value Class</td>
</tr>
<tr>
<td>--------------</td>
<td>----------------------</td>
</tr>
<tr>
<td>1 0 0 0 1</td>
<td>Quiet NaN</td>
</tr>
<tr>
<td>0 1 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
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 7.6.1.12 through 7.6.1.13.

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 2exponent. Encodings are provided in the data format to represent finite numeric values, ±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. 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 (VXISI)
  Infinity+Infinity (VXIDI)
  Zero×Zero (VXZDZ)
  Infinity×Zero (VXIZO)
  Invalid Compare (VXVC)
  Software-Defined Condition (VXSQRT)
  Invalid Square Root (VXSOFT)
  Invalid Integer Convert (VXCVI)

– 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 367 for a description of these exception and enable bits, and Section 7.3.3, “VSX Floating-Point Execution Models” on page 384 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 three different binary fixed-length formats, 16-bit half-precision, 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 data in storage and registers. The single-precision format is used for single-precision data in storage and registers. The double-precision format is used for double-precision 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 three formats. The structure of the half-precision, single-precision, double-precision, and quad-precision formats is shown below.

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. Floating-point half-precision format](image)

![Figure 110. Floating-point single-precision format](image)

![Figure 111. Floating-point double-precision format](image)

![Figure 112. Floating-point quad-precision format (binary128)](image)
<table>
<thead>
<tr>
<th></th>
<th>binary16</th>
<th>binary32</th>
<th>binary64</th>
<th>binary128</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exponent Bias</td>
<td>+15</td>
<td>+127</td>
<td>+1023</td>
<td>+16383</td>
</tr>
<tr>
<td>Maximum Exponent</td>
<td>+15</td>
<td>+127</td>
<td>+1023</td>
<td>+16383</td>
</tr>
<tr>
<td>Minimum Exponent</td>
<td>-14</td>
<td>-126</td>
<td>-1022</td>
<td>-16382</td>
</tr>
<tr>
<td>Widths (bits):</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Format</td>
<td>16</td>
<td>32</td>
<td>64</td>
<td>128</td>
</tr>
<tr>
<td>Sign</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>15</td>
</tr>
<tr>
<td>Exponent</td>
<td>5</td>
<td>8</td>
<td>11</td>
<td>112</td>
</tr>
<tr>
<td>Fraction</td>
<td>10</td>
<td>23</td>
<td>52</td>
<td>113</td>
</tr>
<tr>
<td>Significand</td>
<td>11</td>
<td>24</td>
<td>53</td>
<td></td>
</tr>
</tbody>
</table>

| Nmax             | \((2^{-10}) \times 2^{15} \times 6.6 \times 10^4\) | \((1.2^{-24}) \times 2^{128} \times 3.4 \times 10^{38}\) | \((1.2^{-53}) \times 2^{1024} \times 1.8 \times 10^{308}\) | \((1.2^{-113}) \times 2^{16384} \times 1.2 \times 10^{4932}\) |
| Nmin             | \(1.0 \times 2^{-14} \times 6.1 \times 10^5\) | \(1.0 \times 2^{-126} \times 1.2 \times 10^{-38}\) | \(1.0 \times 2^{-1022} \times 2.2 \times 10^{-368}\) | \(1.0 \times 2^{-16382} \times 3.4 \times 10^{-4932}\) |
| Dmin             | \(1.0 \times 2^{-24} \times 6.0 \times 10^8\) | \(1.0 \times 2^{-144} \times 1.4 \times 10^{-45}\) | \(1.0 \times 2^{-1074} \times 4.9 \times 10^{-324}\) | \(1.0 \times 2^{-16494} \times 6.5 \times 10^{-4966}\) |

Nmax, Nmin, Dmin - Smallest (in magnitude) representable denormalized number.

Value is approximate.

Table 3. IEEE 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 113.

Figure 113. Approximation to real numbers

| -INF | -NOR | -DEN | +0  | +DEN | +NOR | +INF |

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 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:

\[ \text{NOR} = (-1)^{s} \times 2^{E} \times (1.\text{fraction}) \]

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.

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:

\[ \text{DEN} = (-1)^{s} x 2^{E_{\min}} x (0.\text{fraction}) \]

where \( E_{\min} \) is the minimum representable exponent value.

-14 for half-precision  
-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 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 390.

For comparison operations, +\( +\text{Infinity} \) compares equal to +\( +\text{Infinity} \) and -\( -\text{Infinity} \) compares equal to -\( -\text{Infinity} \).

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.

\[ f(\text{src1}, \text{src3}, \text{src2}) \]

ex: \( \text{result} = (\text{src1} \times \text{src3}) - \text{src2} \)

\[ f(\text{src1}, \text{src2}) \]

ex: \( \text{result} = \text{src1} \times \text{src2} \)

ex: \( \text{result} = \text{src1} + \text{src2} \)

\[ f(\text{src1}) \]

ex: \( \text{result} = f(\text{src1}) \)

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.

\[
\text{if src1 is a NaN}
\]

\[
\text{then result = Quiet(src1)}
\]

\[
\text{else if src2 is a NaN (if there is a src2)}
\]

\[
\text{then result = Quiet(src2)}
\]

\[
\text{else if src3 is a NaN (if there is a src3)}
\]

\[
\text{then result = Quiet(src3)}
\]

\[
\text{else if disabled invalid operation exception}
\]

\[
\text{then result = generated QNaN}
\]

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,

\( 0x7FC0_0000 \) for single-precision results,

\( 0x7FF8_0000_0000_0000_0000_0000_0000_0000 \) for double-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 \(-\infty\), 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 \( \text{Square Root} \) or \( \text{Reciprocal Square Root Estimate} \) 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 \( \text{Convert From Integer} \) or \( \text{Round to Floating-Point Integer} \) operation is the sign of the operand being converted.

For the \( \text{Multiply-Add} \) 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 384) 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 409. 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 409) 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 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.

Single-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 Half-Precision to Single-Precision format XX2-form

The half-precision 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 Half-Precision format XX2-form

The single-precision floating-point value in each word element $i$ of the source VSR is rounded to half-precision and placed into the rightmost halfword of the corresponding word element of the target VSR in half-precision 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 instruction.

1. Load Scalar Single-Precision

This form of instruction accesses 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. Scalar Round to Single-Precision

*xrsrp* 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 *xrsrp*, *xrsrp* does not alter the value. Values greater in magnitude than $2^{319}$ when Overflow is enabled ($OE=1$) 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 ($UE=1$) produce undefined results because the value cannot be scaled back into the normalized range.

3. Scalar Convert Single-Precision to Double-Precision

*xscvspdp* accesses a floating-point operand in single-precision format from word element 0 of the source VSR, converts it to double-precision format, and places it into doubleword element 0 of the target VSR.

4. Scalar Convert Double-Precision to Single-Precision

*xscvdpsp* rounds the double-precision floating-point value in doubleword element 0 of the source VSR to single-precision, and places the result into word element 0 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 ($OE=1$) 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 ($UE=1$) produce undefined results because the value cannot be scaled back into the normalized range.

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 OE=1 or UE=1, the result placed into the target VSR and the setting of status bits in the FPSCR are undefined.

For xsresp or xsrsqrtesp, 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

stxsspx converts 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 stxsspx. (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 (lxsspx), a VSX Scalar Round to Single-Precision (xsrsp), or a VSX Scalar Single-Precision Arithmetic instruction is stored in a VSR, the low-order 29 bits of FRACTION are zero.

---

**Programming Note**

VSX Scalar Round to Single-Precision (xsrsp) is provided to allow value conversion from double-precision to single-precision with appropriate exception checking and rounding. xsrsp should be used to convert double-precision floating-point values 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 an xsrsp.

---

**Programming Note**

A single-precision value can be used in double-precision scalar arithmetic operations.

Except for xsresp or xsrsqrtesp, any double-precision value can be used in single-precision scalar arithmetic operations when OE=0 and UE=0. When OE=1 or UE=1, or if the instruction is xsresp or xsrsqrtesp, source operands must be representable in single-precision 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.

---

**Programming Note**

Both single-precision and double-precision forms are provided for most scalar floating-point instructions. Some scalar floating-point instructions are only provided in double-precision form since their operation is identical to the equivalent scalar single-precision operation.

Of the operations for which only a double-precision form of the instruction is provided,

- instructions that return the absolute value, the negative absolute value, or the negated value (xsnabsdp, xsabsdp, xsnegdp) can be used to perform these operations on scalar single-precision operands,

- instructions that perform a comparison (xscmpodp, xscmpudp) can be used to perform these operations on scalar single-precision operands,

- instructions that determine the maximum (xsmaxdp) or minimum (xsmindp) can be used to perform these operations on scalar single-precision operands, and

- instructions that perform an extraction or insertion of the exponent or significand (xscmexpdp, xsieexpdp, xststdcdp, xststdsp, xsxexpdp, xsxsigdp) can be used to perform these operations on scalar single-precision operands.

---

1. VSX Scalar Single-Precision Arithmetic instructions:
xsaddsp, xsdivsp, xsmulsp, xsresp, xssubsp, xsmaddsp, xsmaddmsp, xsmsubsp, xsmsubmsp, xsmsubmsp, xsnmsubasp, xsnmsubmsp
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 \texttt{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\textsuperscript{[1]} 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\textsuperscript{[2]} instructions round each double-precision vector operand element to an integer value in double-precision format.

   VSX Vector Round to Single-Precision Integer\textsuperscript{[3]} 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\textsuperscript{[4]} instructions can cause Invalid Operation (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\textsuperscript{[5]} 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\textsuperscript{[6]} 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\textsuperscript{[7]} 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.

   Rounding is performed using Round Towards Zero rounding mode. These instructions can cause Invalid Operation (VXSNAN, VXCVI) and Inexact exceptions.

3. Converting integer format to floating-point format

   VSX Scalar Integer Doubleword to Double-Precision Format Conversion\textsuperscript{[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\textsuperscript{[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**[1] 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 Word to Double-Precision Format Conversion**[2] 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 Doubleword to Single-Precision Format Conversion**[3] instructions converts the 64-bit signed or unsigned integer in each word element in the source vector operand to double-precision floating-point format.

**VSX Vector Integer Word to Single-Precision Format Conversion**[4] instructions convert the 64-bit signed or unsigned integer in each word element in the source vector operand to single-precision floating-point format.

**VSX Vector Integer Doubleword to Single-Precision Format Conversion**[5] instructions convert the 64-bit signed or unsigned integer to a single-precision floating-point value and returns the result in double-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 increasing 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 round to double-precision integer instructions is normalized and put in double-precision format, and, for the convert double-precision to integer instructions, is converted to a signed or unsigned integer.

**VSX Vector Integer Word to Double-Precision Format Conversion** converts a 64-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 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 increasing 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 round to double-precision integer instructions is normalized and put in double-precision format, and, for the convert double-precision to integer instructions, is converted to a signed or unsigned integer.

The vector round to single-precision integer and vector convert single-precision to integer instructions with biased exponents ranging from 126 through 178 are prepared for rounding by repetitively shifting the significand right one position and increasing the biased exponent until it reaches a value of 179. (Intermediate results with biased exponents 179 or larger are already integers, and with biased exponents 125 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.

FR and 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 FR and FI. If the fraction is incremented during rounding, FR is set to 1, otherwise FR is set to 0. If the result is inexact, FI is set to 1, otherwise FI is set to zero. The scalar round to double-precision integer instructions are exceptions to this rule, setting FR and FI to 0. The scalar double-precision estimate instructions set FR and FI to undefined values. The remaining scalar floating-point instructions do not alter FR and FI.

---

10. VSX Scalar Integer Doubleword to Single-Precision Format Conversion instructions:
xscvsxdisp, xscvuxdisp
1. VSX Vector Integer Doubleword to Double-Precision Format Conversion instructions:
xscvsxddp, xscvuxddp
2. VSX Vector Integer Word to Double-Precision Format Conversion instructions:
xscvsxwddp, xscvuxwddp
3. VSX Vector Integer Doubleword to Single-Precision Format Conversion instructions:
xscvsxdisp, xscvuxdisp
4. VSX Vector Integer Word to Single-Precision Format Conversion instructions:
xscsvsxwsp, xscvuxwsp
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 367. 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 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 430), Round to Nearest Away.

A sixth rounding mode is provided in the quad-precision floating-point instructions, Round to Odd.

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,

```assembly
xsaddqpo Tx,A,B ; use Round to Odd override (RO=1)
xsrqpp Tdp,Tx  ; final QP result rounded to DXP
```

To return a quad-precision result rounded to double-precision requires a 3-instruction sequence,

```assembly
xsaddqpo Tx,A,B ; use Round to Odd override (RO=1)
xcvqpdq Temp,Tx ; DP result rounded & converted to DP
xscvdpqp Tdp,Temp ; final QP result rounded to DP
```

To return a quad-precision result rounded to single-precision requires a 4-instruction sequence,

```assembly
xsaddqpo Tx,A,B ; use Round to Odd override (RO=1)
xcvqpdq Temp,Tx ; DP result rounded to DP using Round to Odd & converted to DP format
xsrsp Temp,Temp ; DP result is rounded to SP
xscvdpqp Tsp,Temp ; final QP result rounded to SP
```

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 Z1 and Z2 bound Z as the next larger and next smaller numbers representable in the target format. Then Z1 or Z2 can be used to approximate the result in the target format.

Figure 114 shows the relation of Z, Z1, and Z2 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 384 for a detailed explanation of rounding.

Figure 114 also summarizes the rounding actions for floating-point intermediate result for all supported rounding modes.
Round to Nearest Away
Choose $Z$ if $Z$ is representable in the target precision.

Otherwise, choose the value that is closer to $Z$ ($Z_1$ or $Z_2$). 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 $Z$ ($Z_1$ or $Z_2$). 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 ($Z_1$ or $Z_2$) 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 ($Z_1$ or $Z_2$).

Round toward $+\infty$
Choose $Z$ if $Z$ is representable in the target precision.

Otherwise, choose $Z_1$.

Round toward $-\infty$
Choose $Z$ if $Z$ is representable in the target precision.

Otherwise, choose $Z_2$. 

Figure 114. Selection of $Z_1$ and $Z_2$
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.
- Underflow 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).

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 double-precision and single-precision floating-point numbers relative to the accumulator illustrated in Figures 109, 110, 111, and 112.

<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

Six rounding modes are provided as described in Section 7.3.2.6, “Rounding” on page 381. The rules for rounding in each mode are as follows.

- **Round to Nearest Even**
  - If IR is exact, choose IR.
  - Otherwise, if G=0, choose NL.
  - Otherwise, if G=1, choose NH.

- **Round to Nearest Away**
  - If IR is exact, choose IR.
  - Otherwise, if G=0, choose NL.
  - Otherwise, if G=1, choose NH.

- **Round towards Zero**
  - If IR is exact, choose IR.
  - Otherwise, choose NL.

- **Round towards +Infinity**
  - If IR is exact, choose IR.
  - Otherwise, if G=1, R=1, or X=1, the least-significant bit of the result is set to 1.

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 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 430).

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 119. 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×Infinity
  - Zero×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 388), 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.

### FE0 FE1 Description

<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 at or beyond 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 might have been used by or might 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 for it to identify the excepting instruction, the operands, and correct the result. No results produced by the excepting instruction have been used by or 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 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.

<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>

### 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.

### FE0 FE1 Description

<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 at or beyond 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 might have been used by or might 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 for it to identify the excepting instruction, the operands, and correct the result. No results produced by the excepting instruction have been used by or 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 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 **VSX Scalar Floating-Point Arithmetic**, **VSX Scalar DP-SP Conversion**, **VSX Scalar Convert Floating-Point to Integer**, and **VSX Scalar Round to Floating-Point Integer** instructions:

1. One or two of the following Invalid Operation exceptions are set to 1.
   
   - `VXSNAN` (if SNaN)
   - `VXSI` (if Infinity–Infinity)
   - `VXOI` (if Infinity±Infinity)
   - `VX2DZ` (if Zero±Zero)
   - `VXI MZ` (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:**
  - `xsmaddqp[r]`, `xsmsubqp[r]`, `xsmnaddqp[r]`, `xsmnsubqp[r]`

**VSX Scalar Quad-Precision Convert to Integer instructions:**
- `xscvqpsdz`, `xscvqpswz`, `xscvqpdz`, `xscvqpuwz`

**VSX Scalar Round Quad-Precision to Double-Extended-Precision (xsrqpxp)**
**VSX Scalar Round to Quad-Precision Integer (xsrqpl)**
**VSX Scalar Convert with round Quad-Precision to Double-Precision format [using round to Odd] (xscvqdpd[r])**

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)
   - VXMZ (if Infinity × Zero)
   - VXSO (if Invalid Square Root)
   - VXCVI (if Invalid Integer Convert)

2. VSR[VT+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 (xscvhpdp)**
**VSX Scalar Convert with round Double-Precision to Half-Precision format (xscvdpdp)**

do the following.

1. VXSNAN is set to 1.
2. VSR[XT] is not modified.
3. FR and FI are set to 0. FPRF is not modified.

For any of the following instructions,

**VSX Vector Convert Half-Precision to Single-Precision format (xcvhpdp)**
**VSX Vector Convert with round Single-Precision to Half-Precision format (xcvphpd)**
do the following.

1. VXSNAN is set to 1.
2. VSR[XT] is not modified.
3. FR, FI, and 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.

   VXSNAN (if SNaN)
   VXISI (if Infinity – Infinity)
   VXIDI (if Infinity ÷ Infinity)
   VXZDZ (if Zero ÷ Zero)
   VXIMZ (if Infinity × Zero)
   VXVC (if Invalid Compare)
   VXSQRT (if Invalid Square Root)
   VXCVI (if Invalid Integer Convert)

2. Update of VSR[XT] is suppressed for all vector elements.
3. FR and FI are unchanged.
4. 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 element 0 of VSR[XT]. The contents of word elements 1-3 of VSR[XT] are undefined.
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)
   - VXISI (if Infinity – Infinity)
   - VXIDI (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].

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 (xscvsdpd) 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)
   - VXISI (if Infinity – Infinity)
   - VXIDI (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 undefined.

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:**
- xsaddqp[o]
- xsdivqp[o]
- xsmulpq[o]
- xssqrtqp[o]
- xssubqp[o]
- xsaddqdp[o]
- xsmulpq[o]
- xsdaddqdp[o]
- xssubqdp[o]

**VSX Scalar Quad-Precision Round to Integer (xsrqpi)**

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)

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 \((\text{xrsqpxp})\), do the following.

1. \(\text{VXSNAN}\) is set to 1.
2. The Quiet NaN is placed into \(\text{VSR}[VRT+32]\) in quad-precision format.
3. \(\text{FR}\) and \(\text{FI}\) are set to 0. \(\text{FPRF}\) is set to indicate the class of the result (Quiet NaN).

For any of the following instructions,

\(\text{VSX Scalar Compare Ordered Quad-Precision (xscmpoqp)}\)
\(\text{VSX Scalar Compare Unordered Quad-Precision (xscmpoqp)}\)

do the following.

1. One or two of the following Invalid Operation exceptions are set to 1.

\[\text{VXSNAN} \quad \text{(if SNaN)}\]
\[\text{VXVC} \quad \text{(if Invalid Compare)}\]

2. \(\text{FR}\), \(\text{FI}\) and \(\text{C}\) are unchanged. \(\text{FPCC}\) is set to reflect unordered.

For VSX Scalar Convert with round Quad-Precision to Double-Precision format [using round to Odd] \((\text{xscvpdp}[o])\), do the following.

1. \(\text{VXSNAN}\) is set to 1.
2. The double-precision Quiet NaN result is placed into doubleword element 0 of \(\text{VSR}[VRT+32]\) in double-precision format.

\[0x0000_0000_0000_0000\] is placed into doubleword element 1 of \(\text{VSR}[VRT+32]\).

3. \(\text{FR}\) and \(\text{FI}\) are set to 0. \(\text{FPRF}\) 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 \((\text{xscvqpsz})\), do the following.

1. One or two of the following Invalid Operation exceptions are set to 1.

\[\text{VXSNAN} \quad \text{(if SNaN)}\]
\[\text{VXCVI} \quad \text{(if Invalid Integer Convert)}\]

2. \(0x\text{FFFF_FFFF_FFFF_FFFF}\) is placed into doubleword element 0 of \(\text{VSR}[VRT+32]\) if the quad-precision operand in \(\text{VSR}[VRB+32]\) is a positive number or +Infinity.

\(0x\text{8000_0000_0000_0000}\) is placed into doubleword element 0 of \(\text{VSR}[VRT+32]\) if the quad-precision operand in \(\text{VSR}[VRB+32]\) is a negative number, −Infinity, or NaN.

\(0x\text{0000_0000_0000_0000}\) is placed into doubleword element 1 of \(\text{VSR}[VRT+32]\).

3. \(\text{FR}\) and \(\text{FI}\) are set to 0. \(\text{FPRF}\) is undefined.

For VSX Scalar Convert with round to zero Quad-Precision to Signed Word format \((\text{xscvqpswz})\), do the following.

1. One or two of the following Invalid Operation exceptions are set to 1.

\[\text{VXSNAN} \quad \text{(if SNaN)}\]
Chapter 7. Vector-Scalar Floating-Point Operations

VXCVI  (if Invalid Integer Convert)

2. $0x7FFF_FFFF$ is placed into word element 1 of $\text{VSR}[\text{VRT}+32]$ if the quad-precision operand in $\text{VSR}[\text{VRB}+32]$ is a positive number or $+\infty$.

$0x8000_0000$ is placed into word element 1 of $\text{VSR}[\text{VRT}+32]$ if the quad-precision operand in $\text{VSR}[\text{VRB}+32]$ is a negative number, $-\infty$, or NaN.

$0x0000_0000$ is placed into word elements 0, 2, and 3 of $\text{VSR}[\text{VRT}+32]$.

3. $\text{FR}$ and $\text{FI}$ are set to 0. $\text{FPRF}$ is undefined.

For \textit{VSX Scalar Convert with round to zero Quad-Precision to Unsigned Doubleword format} ($\text{xscvqpdz}$), 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{VXCVI} & \quad \text{(if Invalid Integer Convert)}
\end{align*}

2. $0xFFFF_FFFF$ is placed into doubleword element 0 of $\text{VSR}[\text{VRT}+32]$ if the quad-precision operand in $\text{VSR}[\text{VRB}+32]$ is a positive number or $+\infty$.

$0x0000_0000_0000_0000$ is placed into doubleword element 0 of $\text{VSR}[\text{VRT}+32]$ if the quad-precision operand in $\text{VSR}[\text{VRB}+32]$ is a negative number, $-\infty$, or NaN.

$0x0000_0000_0000_0000$ is placed into doubleword element 1 of $\text{VSR}[\text{VRT}+32]$.

3. $\text{FR}$ and $\text{FI}$ are set to 0. $\text{FPRF}$ is undefined.

For \textit{VSX Scalar Convert with round to zero Quad-Precision to Unsigned Word format} ($\text{xscvpuwz}$), 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{VXCVI} & \quad \text{(if Invalid Integer Convert)}
\end{align*}

2. $0xFFFF_FFFF$ is placed into word element 1 of $\text{VSR}[\text{VRT}+32]$ if the quad-precision operand in $\text{VSR}[\text{VRB}+32]$ is a positive number or $+\infty$.

$0x0000_0000$ is placed into word element 1 of $\text{VSR}[\text{VRT}+32]$ if the quad-precision operand in $\text{VSR}[\text{VRB}+32]$ is a negative number, $-\infty$, or NaN.

$0x0000_0000$ is placed into word elements 0, 2, and 3 of $\text{VSR}[\text{VRT}+32]$.

3. $\text{FR}$ and $\text{FI}$ are set to 0. $\text{FPRF}$ is undefined.

For \textit{VSX Scalar Convert with round Double-Precision to Half-Precision format} ($\text{xscvdphp}$), do the following.

1. $\text{VXSNAN}$ is set to 1.

2. The half-precision representation of a Quiet NaN is placed into the rightmost halfword of doubleword element 0 of $\text{VSR}[\text{XT}]$. 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 undefined.

3. $\text{FR}$ and $\text{FI}$ are set to 0. $\text{FPRF}$ is set to indicate the class of the result (Quiet NaN).
For **VSX Scalar Convert Half-Precision to Double-Precision format (xscvhdpdp)**, do the following.

1. VXSNAN is set to 1.
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 undefined.
3. FR and FI are set to 0. FPRF is set to indicate the class of the result (Quiet NaN).

For the **VSX Vector Double-Precision Arithmetic instructions**, **VSX Vector Double-Precision Maximum/Minimum instructions**, the **VSX Vector Convert Single-Precision to Double-Precision format (xvcvspdp)** instruction, and the **VSX Vector Round to Double-Precision Integer instructions**:

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)

2. The double-precision representation of a Quiet NaN is placed into its respective doubleword element of VSR[XT].
3. FR, FI, and FPRF are not modified.

For the **VSX Scalar Convert with round to zero Double-Precision to Signed Doubleword format (xscvdpsxd)** instruction:

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[XT] if the double-precision operand in doubleword element 0 of VSR[XB] is a positive number or +Infinity.

   0x8000_0000_0000_0000 is placed into doubleword element 0 of VSR[XT] if the double-precision operand in doubleword element 0 of VSR[XB] is a negative number, –Infinity, or NaN.

   The contents of doubleword element 1 of VSR[XT] are undefined.
3. FR and FI are set to 0.
4. FPRF is undefined.

For the **VSX Scalar Convert with round to zero Double-Precision to Unsigned Doubleword format (xscvdpuxd)** instruction:

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[XT] if the double-precision operand in doubleword element 0 of VSR[XB] is a positive number or +Infinity.

   0x0000_0000_0000_0000 is placed into doubleword element 0 of VSR[XT] if the double-precision operand in doubleword element 0 of VSR[XB] is a negative number, –Infinity, or NaN.
The contents of doubleword element 1 of VSR[XT] are undefined.

3. 
FR and FI are set to 0.

4. 
FPRF is undefined.

For the **VSX Scalar Convert with round to zero Double-Precision to Signed Word format (xscvdpfxw)** instruction:

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[XT] if the double-precision operand in doubleword element 0 of VSR[XB] is a positive number or +Infinity.

   0x8000_0000 is placed into word element 1 of VSR[XT] if the double-precision operand in doubleword element 0 of VSR[XB] is a negative number, –Infinity, or NaN.

The contents of word elements 0, 2, and 3 of VSR[XT] are undefined.

3. 
FR and FI are set to 0.

4. 
FPRF is undefined.

For the **VSX Scalar Convert with round to zero Double-Precision to Unsigned Word format (xscvdpuxw)** instruction:

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 element 1 of VSR[XT] if the double-precision operand in doubleword element 0 of VSR[XB] is a positive number or +Infinity.

   0x0000_0000 is placed into word element 1 of VSR[XT] if the double-precision operand in doubleword element 0 of VSR[XB] is a negative number, –Infinity, or NaN.

The contents of word elements 0, 2, and 3 of VSR[XT] are undefined.

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 (xvcvdpfxd)** instruction:

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 +Infinity.
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:

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 \(+\infty\).

   \( 0x0000_0000_0000_0000 \) 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** (**xvcvdpuxw**) instruction:

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 word element \( i \times 2 \) of VSR[XT] if the double-precision operand in doubleword element \( i \) of VSR[XB] is a positive number or \(+\infty\).

   \( 0x8000_0000 \) is placed into word element \( i \times 2 \) of VSR[XT] if the double-precision operand in doubleword element \( i \) of VSR[XB] is a negative number, \(-\infty\), or NaN.

   The contents of word element \( i \times 2 + 1 \) of VSR[XT] are undefined.

3. FR, FI, and FPRF are not modified.

For the **VSX Vector Convert with round to zero Double-Precision to Unsigned Word format** (**xvcvdpuxw**) instruction:

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 word element \( i \times 2 \) of VSR[XT] if the double-precision operand in doubleword element \( i \) of VSR[XB] is a positive number or \(+\infty\).

   \( 0x0000_0000 \) is placed into word element \( i \times 2 \) of VSR[XT] if the double-precision operand in doubleword element \( i \) of VSR[XB] is a negative number, \(-\infty\), or NaN.

   The contents of word element \( i \times 2 + 1 \) of VSR[XT] are undefined.

3. FR, FI, and FPRF are not modified.
For the VSX Vector Convert with round to zero Single-Precision to Signed Doubleword format ($xvcvspsxd$) instruction:

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 single-precision operand in word element $i \times 2$ of VSR[XB] is a positive number or $+\text{Infinity}$.
   $0x8000_0000_0000_0000$ is placed into doubleword element $i$ of VSR[XT] if the single-precision operand in word element $i \times 2$ of VSR[XB] is a negative number, $-\text{Infinity}$, or NaN.

3. FR, FI, and FPRF are not modified.

For the VSX Vector Convert with round to zero Single-Precision to Unsigned Doubleword format ($xvcvspuxd$) instruction:

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 single-precision operand in word element $i \times 2$ of VSR[XB] is a positive number or $+\text{Infinity}$.
   $0x0000_0000_0000_0000$ is placed into doubleword element $i$ of VSR[XT] if the single-precision operand in word element $i \times 2$ of VSR[XB] is a negative number, $-\text{Infinity}$, or NaN.

3. FR, FI, and FPRF are not modified.

For the VSX Vector Convert with round to zero Single-Precision to Signed Word format ($xvcvspsxw$) instruction:

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 $i$ of VSR[XT] if the single-precision operand in word element $i$ of VSR[XB] is a positive number or $+\text{Infinity}$.
   $0x8000_0000$ is placed into word element $i$ of VSR[XT] if the single-precision operand in word element $i$ of VSR[XB] is a negative number, $-\text{Infinity}$, or NaN.

   The contents of word element $2i + 1$ of VSR[XT] are undefined.

3. FR, FI, and FPRF are not modified.

For the VSX Vector Convert with round to zero Single-Precision to Unsigned Word format ($xvcvspuxw$) instruction:

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 element $i$ of VSR[XT] if the single-precision operand in the corresponding word element $2i$ of VSR[XB] is a positive number or $+\text{Infinity}$. 
0x0000_0000 is placed into word element \( i \) of VSR[XT] if the single-precision operand in word element \( 2xi \) of VSR[XB] is a negative number, -Infinity, or NaN.

The contents of word element \( 2xi + 2 \) of VSR[XT] are undefined.

3. \( FR,FI, \) and \( FPRF \) are not modified.

For the VSX Scalar Floating-Point Compare instructions:

1. One or two of the following Invalid Operation exceptions are set to 1.

\[
\text{\texttt{VXSNAN}} \quad \text{(if SNaN)}
\]
\[
\text{\texttt{VXCVI}} \quad \text{(if Invalid Integer Convert)}
\]

2. \( FR,FI, \) and \( C \) are unchanged.

3. \( FPCC \) is set to reflect unordered.

For the VSX Vector Compare Single-Precision instructions:

1. One or two of the following Invalid Operation exceptions are set to 1.

\[
\text{\texttt{VXSNAN}} \quad \text{(if SNaN)}
\]
\[
\text{\texttt{VXCVI}} \quad \text{(if Invalid Integer Convert)}
\]

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:

1. One or two of the following Invalid Operation exceptions are set to 1.

\[
\text{\texttt{VXSNAN}} \quad \text{(if SNaN)}
\]
\[
\text{\texttt{VXCVI}} \quad \text{(if Invalid Integer Convert)}
\]

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 (\texttt{xscvphp}), 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 (\texttt{xscvhpsp}), 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\(^1\) 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\(^2\) instruction or a VSX Floating-Point Reciprocal Square Root Estimate\(^3\) 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 Floating-Point Divide instructions: \texttt{xsdivdp}, \texttt{xsdivsp}
- VSX Scalar Floating-Point Reciprocal Estimate instructions: \texttt{xsredp}, \texttt{xresp}
- VSX Scalar Floating-Point Reciprocal Square Root Estimate instructions: \texttt{xsrsqrtedp}, \texttt{xsrsqrtesp}

do the following.

1. \( \text{ZE} \) is set to 1.
2. Update of \( \text{VSR}[\text{XT}] \) is suppressed.
3. \( \text{FR} \) and \( \text{FI} \) are set to 0.
4. \( \text{FPRF} \) is unchanged.

For VSX Scalar Divide Quad-Precision (\texttt{xsdivqp}), do the following.

1. \( \text{ZE} \) is set to 1.
2. Update of \( \text{VSR}[\text{VRT}+32] \) is suppressed.
3. \( \text{FR} \) and \( \text{FI} \) are set to 0. \( \text{FPRF} \) is not modified.

For any of the following instructions,

- VSX Vector Floating-Point Divide instructions: \texttt{xsdivdp}, \texttt{xsdivsp}, \texttt{xvdivdp}, \texttt{xdivdp}
- VSX Vector Floating-Point Reciprocal Estimate instructions: \texttt{xsredp}, \texttt{xresp}, \texttt{xvredp}, \texttt{xvresp}
- VSX Vector Floating-Point Reciprocal Square Root Estimate instructions: \texttt{xsrsqrtedp}, \texttt{xsrsqrtesp}, \texttt{xvrsqrtedp}, \texttt{xvrsqrtesp}

---

1. VSX Vector Floating-Point Divide instructions: \texttt{xsdivdp}, \texttt{xsdivsp}, \texttt{xvdivdp}, \texttt{xdivdp}
2. VSX Floating-Point Reciprocal Estimate instructions: \texttt{xsredp}, \texttt{xresp}, \texttt{xvredp}, \texttt{xvresp}
3. VSX Floating-Point Reciprocal Square Root Estimate instructions: \texttt{xsrsqrtedp}, \texttt{xsrsqrtesp}, \texttt{xvrsqrtedp}, \texttt{xvrsqrtesp}
do the following.

1. \( ZX \) is set to 1.
2. Update of \( VSR[XT] \) is suppressed for all vector elements.
3. \( FR \) and \( FI \) are unchanged.
4. \( FPRF \) 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 VSX Scalar Floating-Point Divide\(^1\) instructions, do the following.

1. \( ZX \) 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 undefined.
3. \( FR \) and \( FI \) are set to 0.
4. \( FPRF \) is set to indicate the class and sign of the result (\( \pm \) Infinity).

For VSX Scalar Divide Quad-Precision (xsdivq), do the following.

1. \( ZX \) 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 (\( \pm \) Infinity).

For VSX Vector Divide Double-Precision (xvdvdp), do the following.

1. \( ZX \) 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 (xvdvsp), do the following.

1. \( ZX \) 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.

---

1. VSX Scalar Floating-Point Divide instructions: xsdivdp, xsdivsp
For **VSX Scalar Floating-Point Reciprocal Estimate** instructions and **VSX Scalar Floating-Point Reciprocal Square Root Estimate** instructions, 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 undefined.

3. **FR** and **FI** are set to 0.

4. **FPRF** is set to indicate the class and sign of the result (± Infinity).

For the **VSX Vector Reciprocal Estimate Double-Precision** (xvredp) and **VSX Vector Reciprocal Square Root Estimate Double-Precision** (xvrsqrtdp) instructions:

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 the **VSX Vector Reciprocal Estimate Single-Precision** (xvresp) and **VSX Vector Reciprocal Square Root Estimate Single-Precision** (xvrsqrtesp) instructions:

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.

---

1. VSX Scalar Floating-Point Reciprocal Estimate instructions:
   - xsredp, xsresp
2. VSX Scalar Floating-Point Reciprocal Square Root Estimate instructions:
   - xsrsqrtdp, xsrsqrtesp
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 (xscvdp) 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 element 0 of VSR[XT] in single-precision format. The contents of word elements 1-3 of VSR[XT] are undefined.
4. Unless the result is undefined, FPRF is set to indicate the class and sign of the result (±Normal Number).

For VSX Scalar Double-Precision Arithmetic\(^1\) instructions, 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 undefined.
4. FPRF is set to indicate the class and sign of the result (±Normal Number).

For VSX Scalar Single-Precision Arithmetic\(^2\) instructions, 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 undefined.
4. FPRF is set to indicate the class and sign of the result (±Normal Number).

---

1. VSX Scalar Double-Precision Arithmetic instructions:
   - xsaddp, xsdivdp, xsmulp, xsredp, xssubdp, xsmadadp, xsmaddmdp, xsmsubadp, xsmsubmdp, xsnmaddadp, xsnmaddmdp, xsnmmsubadp, xsnmmsubmdp

2. VSX Scalar Single-Precision Arithmetic instructions:
   - xsadsp, xsdivsp, xsmuls, xssredsp, xssubsp, xsmsaddsp, xsmaddmsp, xsmsubsp, xsmsubmsp, xsnmaddasp, xsnmaddmsp, xsnmmsubasp, xsnmmsubmsp
For any of the following instruction classes,

**VSX Scalar Quad-Precision Arithmetic instructions:**
- `xsmaddqp[o]`, `xsmsubqp[o]`, `xsnmaddqp[o]`, `xsnmsubqp[o]`

**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] (**xscvqdpd**), 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 (\( E_{max} \)), 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 undefined.
4. Unless the result is undefined, \( FPRF \) is set to indicate the class and sign of the result (±Normal Number).
For VSX Vector Double-Precision Arithmetic\textsuperscript{1} instructions, VSX Vector Single-Precision Arithmetic\textsuperscript{2} instructions, and VSX Vector round and Convert Double-Precision to Single-Precision format instruction (\texttt{xvcvdpsp}), do the following.

1. OX is set to 1.
2. Update of VSR[XT] is suppressed for all vector elements.
3. FR, FI, and FPRF are not modified.

For VSX Vector Convert with round Single-Precision to Half-Precision format (\texttt{xvcvsphp}), do the following.

1. OX is set to 1.
2. VSR[XT] is not modified.
3. FR, FI, and FPRF are not modified.

---

\textsuperscript{1} VSX Vector Double-Precision Arithmetic instructions:
- \texttt{xvaddp, xvdivdp, xvmuldp, xvredp, xvsbdp, xvmaddoadp, xsmaddmdp, xvmsubadp, xvmsubmdp, xvmaddadp, xvmaddmdp, xvnmaddadp, xvnmaddmdp, xvnmmsubadp, xvnmmsubmdp}

\textsuperscript{2} VSX Vector Single-Precision Arithmetic instructions:
- \texttt{xvadsp, xvdivsp, xvmuls, xvresp, xvsbsp, xvmaddsp, xvmsubasp, xvmaddmsp, xvmsubmsp, xvnmaddasp, xvnmaddmsp, xvnmmsubasp, xvnmmsubmsp}
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 \(-\text{Infinity}\).  
   For positive overflow, the result is \(+\text{Infinity}\).

   **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 \(+\text{Infinity}\)**  
   For negative overflow, the result is the format's most negative finite number.  
   For positive overflow, the result is \(+\text{Infinity}\).

   **Round toward \(-\text{Infinity}\)**  
   For negative overflow, the result is \(-\text{Infinity}\).  
   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 element 0 of \(VSR[XT]\) as a single-precision value. The contents of word elements 1-3 of \(VSR[XT]\) are undefined.
4. \(FR\) is undefined.
5. \(FI\) is set to 1.
6. \(FPRF\) is set to indicate the class and sign of the result.

For **VSX Scalar Double-Precision Arithmetic**\(^1\) instructions and **VSX Scalar Single-Precision Arithmetic**\(^2\) instructions, 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 undefined.
4. \(FR\) is undefined.
5. \(FI\) is set to 1.
6. \(FPRF\) is set to indicate the class and sign of the result.

For any of the following instructions,

**VSX Scalar Quad-Precision Arithmetic** instructions:

- \(xsaddqp[o]\), \(xsdivqp[o]\), \(xsmulqp[o]\), \(xssubqp[o]\)
- \(xsmaddqp[o]\), \(xsmsubqp[o]\), \(xsnmaddqp[o]\), \(xsnmsubqp[o]\)

**VSX Scalar Quad-Precision Round to Double-Extended-Precision** \((xsrqpxp)\)

---

1. **VSX Scalar Double-Precision Arithmetic instructions:**  
   \(xsaddp, xsdivp, xsmulp, xsrdiv, xsradd, xsraddp, xsmaddp, xsmaddp, xsmsubdp, xsmsubdp, xsnmaddp, xsnmaddp, xsnmsubd, xsnmsubd, xsnmsubd, xsnmsubd\)

2. **VSX Scalar Single-Precision Arithmetic instructions:**  
   \(xssaddp, xssdivp, xssmulp, xssrdiv, xssradd, xssraddp, xssmaddp, xssmaddp, xssmsubdp, xssmsubdp, xssmsubdp, xssmsubdp, xssmsubd, xssmsubd, xssmsubd, xssmsubd\)
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 ($xscvqdpdp$), do the following.

3. The result is placed into doubleword element 0 of $VSR[VRT+32]$ as a double-precision value.

$0\times 0000_0000\_0000\_0000$ 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.

1. $OX$ and $XX$ are 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 undefined.

3. $FR$ is undefined. $FI$ is set to 1. $FPRF$ is set to indicate the class and sign of the result.

For VSX Vector Double-Precision Arithmetic\(^1\) instructions, 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 VSX Vector Single-Precision Arithmetic\(^2\) instructions and 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.

4. $FR$, $FI$, and $FPRF$ are not modified.

For VSX Vector Convert with round Single-Precision to Half-Precision format ($xvcvsphp$), do the following.

1. $OX$ and $XX$ are set to 1.

2. 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.

3. $FR$, $FI$, and $FPRF$ are not modified.

---

1. VSX Vector Double-Precision Arithmetic instructions:
   xvadddp, xvdivdp, xvmuldp, xvreddp, xvsubdp, xvmadaddp, xvmaddmdp, xvmsubadp, xvmsubmdp, xvmaddadp, xvmaddmddp, xvnmaddadp, xvnmaddmdp, xvnmsubadp, xvnmsubmdp

2. VSX Vector Single-Precision Arithmetic instructions:
   xvadsp, xvdivsp, xvmuls, xvrepsp, xvsups, xvmaddsp, xvmaddmsp, xvmsubmsp, xvmaddasp, xvmaddmsp, xvmsubmsp, xvmaddasp, xvmaddmsp, xvnmadgsp, xvnmsubmsp
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 (UE = 0), the intermediate result is denormalized (see Section 7.3.2.4, “Normalization and Denormalization” on page 377) and rounded (see Section 7.3.2.6, “Rounding” on page 381) 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 (UE=1) and an Underflow exception occurs, the following actions are taken:

For **VSX Scalar round and Convert Double-Precision to Single-Precision format (xscvdp)**, do the following.

1. UX is set to 1.
2. If the unbiased exponent of the normalized intermediate result is greater than or equal to -319 (E.min - 192), the exponent is adjusted by adding 192. Otherwise the result is undefined.
3. The adjusted rounded result is placed into word element 0 of VSR[XT] in single-precision format. The contents of word elements 1-3 of VSR[XT] are undefined.
4. Unless the result is undefined, FPRF is set to indicate the class and sign of the result (±Normal Number).

For **VSX Scalar Double-Precision Arithmetic** instructions and 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 doubleword element 0 of VSR[XT] in double-precision format. The contents of doubleword element 1 of VSR[XT] are undefined.
4. FPRF is set to indicate the class and sign of the result (±Normal Number).

---

1. VSX Scalar Double-Precision Arithmetic instructions: xadddp, xsdvpd, xsmuldp, xsubdp, xsmadddp, xsmaddmp, xsmsubdp, xsmsubmdp, xsnmadddp, xsnmaddmp, xsnmsubdp, xsnmsubmdp
For any of the following instructions,

**VSX Scalar Quad-Precision Arithmetic instructions:**
- `xsaddqp`[0], `xsdivqp`[0], `xsmulpq`[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. `UX` is set to 1.
2. The exponent of the normalized intermediate result is adjusted by adding 14576.
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] (xscvqdp)[0]**, 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 exponent is less than -1022, 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 Single-Precision Arithmetic**[1] instructions and **VSX Scalar Single-Precision Reciprocal Estimate (xsresp)**, do the following.

1. `UX` is set to 1.
2. The exponent is adjusted by adding 192.
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 undefined.
4. `FPRF` is set to indicate the class and sign of the result (±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.

---

1. **VSX Scalar Single-Precision Arithmetic instructions:**
- `xsaddsp`, `xsdivsp`, `xsmulsp`, `xssubsp`, `xsmaddsp`, `xsmaddsp`, `xsmsubasp`, `xsnmaddasp`, `xsnmaddasp`, `xsnmsubasp`, `xsnmsubmisp`
2. The exponent of the normalized intermediate result is adjusted by adding $24$. If the adjusted exponent is less than $-14$, 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 undefined.

4. Unless the result is undefined, $FPRF$ is set to indicate the class and sign of the result (±Normal Number).

For VSX Vector Floating-Point Arithmetic$^{[1]}$ instructions, VSX Vector Floating-Point Reciprocal Estimate$^{[2]}$ instructions, and VSX Vector round and Convert Double-Precision to Single-Precision format ($xvcvdpsp$), do the following:

1. $UX$ is set to 1.
2. Update of $VSR[XT]$ is suppressed for all vector elements.
3. $FR$, $FI$, and $FPRF$ are not modified.

For VSX Vector Convert with round Single-Precision to Half-Precision format ($xvcvspdp$), 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 ($xscvdpsp$), do the following.

1. $UX$ is set to 1.
2. The result is placed into word element 0 of $VSR[XT]$ in single-precision format. The contents of word elements 1-3 of $VSR[XT]$ are undefined.
3. $FPRF$ is set to indicate the class and sign of the result.

For VSX Scalar Floating-Point Arithmetic$^{[3]}$ instructions and VSX Scalar Reciprocal Estimate$^{[4]}$ instructions, 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 undefined.
3. 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[o]`, `xsdvqp[o]`, `xsmulqp[o]`, `xssubqp[o]`
- `xsmaddqp[o]`, `xsmsubqp[o]`, `xsnmaddqp[o]`, `xsnmsubqp[o]`

**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 **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 **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 undefined.
3. FPRF is set to indicate the class and sign of the result.

For **VSX Vector Double-Precision Arithmetic**\(^1\) instructions and **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 **VSX Vector Single-Precision Arithmetic**\(^2\), **VSX Vector Reciprocal Estimate Single-Precision (xvresp)**, and **VSX Vector round and Convert Double-Precision to Single-Precision format (xvcvdpsp)**, do the following.

1. UX is set to 1.

---

1. **VSX Vector Double-Precision Arithmetic** instructions:
   - `xvadddp`, `xvdvdp`, `xvmuldp`, `xvsubdp`, `xvmadddp`, `xvmaddmp`, `xvmsubdp`, `xvmsubmp`, `xvnmadddp`, `xvnmaddmp`, `xvnmsubdp`, `xvnmsubmp`

2. **VSX Vector Single-Precision Arithmetic** instructions:
   - `xvaddsp`, `xvdvsp`, `xvmulsp`, `xvsubsp`, `xvmaddsp`, `xvmaddmsp`, `xvmsubsp`, `xvmsubmsp`, `xvnmaddsp`, `xvnmaddmsp`, `xvnmsubasp`, `xvnmsubmsp`
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.

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 VSX Vector round and Convert Double-Precision to Single-Precision format (xscvdpsp) instruction:

1. XX is set to 1.
2. The result is placed into word element 0 of VSR[XT] in single-precision format. The contents of word elements 1-3 of VSR[XT] are undefined.
3. FPRF is set to indicate the class and sign of the result.

For VSX Scalar Floating-Point Arithmetic[1] instructions, VSX Scalar Round to Double-Precision Integer Exact using Current rounding mode (xsrdpic), and VSX Scalar Integer to Floating-Point Format Conversion[2] instructions, do the following.

1. XX 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 undefined.
3. FPRF is set to indicate the class and sign of the result.

For VSX Scalar Floating-Point to Integer Word Format Conversion[3] instructions, do the following.

1. XX is set to 1.
2. The result is placed into word element 1 of VSR[XT]. The contents of word elements 0, 2, and 3 of VSR[XT] are undefined.
3. FPRF is set to indicate the class and sign of the result.

---

1. VSX Scalar Floating-Point Arithmetic instructions:
   - xsadddp, xsdivdp, xsmuldp, xssubdp, xsaddsp, xsdivsp, xsmulsp, xssubsp, xsaddadp, xsaddmdp, xssubadp, xssubmdp, xsnaddadp, xsnaddmdp, xsnsubadp, xsnsubmdp, xsmaddadp, xsmaddmdp, xsmsubadp, xsmsubmdp, xsmadddsp, xsmaddmsp, xsmsubadsp, xsmsubmdsp, xsmadddsp, xsmaddmsp, xsmsubadsp, xsmsubmdsp, xsnadddsp, xsnaddmsp, xsnsubadsp, xsnsubmdsp

2. VSX Scalar Integer to Floating-Point Format Conversion instructions:
   - xscvddp, xscvuxdp, xscvsxdp, xscvuxdp

3. VSX Scalar Floating-Point to Integer Word Format Conversion instructions:
   - xscvdpsxws, xscvdpxws

---
For any of the following instructions,

**VSX Scalar Quad-Precision Arithmetic instructions:**
- `xsmaddqp[o]`, `xsmsubqp[o]`, `xsnmaddqp[o]`, `xsnmsubqp[o]`

**VSX Scalar Quad-Precision Round instructions:**
- `xsrqi`, `xsrqpxp`

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 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 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 **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** (**xscvqpu dz**), 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 (xsvqpuwz), 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 undefined.
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 Vector Floating-Point Arithmetic\(^1\) instructions, VSX Vector Floating-Point Reciprocal Estimate\(^2\) instructions, VSX Vector round and Convert Double-Precision to Single-Precision format (xvcvdpsp), VSX Vector Double-Precision to Integer Format Conversion\(^3\) instructions, and VSX Vector Integer to Floating-Point Format Conversion\(^4\) instructions, do the following.

1. XX is set to 1.
2. Update of VSR[XT] is suppressed for all vector elements.
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. 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 element 0 of VSR[XT] as a single-precision value. The contents of word elements 1-3 of VSR[XT] are undefined.
3. FPRF is set to indicate the class and sign of the result.

For VSX Scalar Double-Precision Arithmetic\(^1\) instructions, VSX Scalar Single-Precision Arithmetic\(^2\) instructions, VSX Scalar Round to Single-Precision (xsrsp), the VSX Scalar Round to Double-Precision Integer Exact using Current rounding mode (xsrdrpc), and VSX Scalar Integer to Double-Precision Format Conversion\(^3\) instructions, 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 undefined.
3. FPRF is set to indicate the class and sign of the result.

For VSX Scalar Convert with round to zero Double-Precision To Signed Word format (xscvdpsxws) and VSX Scalar Convert with round to zero Double-Precision To Unsigned Word format (xscvdpuxws), do the following.

1. XX is set to 1.
2. The result is placed into word element 1 of VSR[XT]. The contents of word elements 0, 2, and 3 of VSR[XT] are undefined.
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 (xscvqdpd), 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 undefined.
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.

---

1. VSX Scalar Double-Precision Arithmetic instructions:
   xsadddp, xssubdp, xsmuldp, xsdivdp, xssqrtsp, xsmaddsp, xsmadddp, xsmudasdp, xsmudasdp, xsmaddsp, xsnmaddsp, xsnmaddwp, xsnmsubwp, xsnmsubmp
2. VSX Scalar Single-Precision Arithmetic instructions:
   xsaddsp, xssubsp, xsmulsp, xsdivsp, xssqrtsp, xsmaddsp, xsmaddmp, xsmudasmp, xsmudasmp, xsnmaddmp, xsnmaddmp, xsnmsubmp, xsnmsubmp
3. VSX Scalar Integer to Double-Precision Format Conversion instructions:
   xscvsxddp, xscvxudp
For VSX Vector Double-Precision Arithmetic instructions, 
* xvadddp, xsadddp, xvmuldp, xdivdp, xsqrtdp, xvmaddadp, xvmaddmdp, xvmsubadp, xvmsubmdp, 
* 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[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**) 
VSX Scalar Round to Quad-Precision Integer (**xsrqpi**) 

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 (**xscvqdp**), 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. **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 \((xscvqpush)\)  
VSX Scalar truncate & Convert Quad-Precision to Unsigned Word \((xscvquwz)\)

do the following.

1. \(XX\) 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 \((xvcvsphp)\), do the following.

1. \(XX\) 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 VSX Vector Single-Precision Arithmetic\(^1\) instructions, 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 word element of \(VSR[XT]\) in single-precision format.
3. \(FR\), \(FI\), and \(FPRF\) are not modified.

---

1. VSX Vector Single-Precision Arithmetic instructions:  
   \(xvaddsp, xvsubsp, xvmsusp, xvdivsp, xvsqrtsp, xvmsadsp, xvmsadmsp, xvmsubsp, xvmsubmsp, xvnmsadsp, xvnmsadmsp, xvnmsubsp, xvnmsubmsp\)
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.

```
short AW[4] = { 0x0001_0203,
                0x0405_0607,
                0x0809_0A0B,
                0x0C0D_0E0F};
```

Figure 120 illustrates the Big-Endian storage image of array AW.

![Big-Endian storage image of array AW](image)

Figure 120. Big-Endian storage image of array AW

Figure 121 illustrates the Little-Endian storage image of array AW.

![Little-Endian storage image of array AW](image)

Figure 121. Little-Endian storage image of array AW

Figure 122 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 120 for Big-Endian. Note that Figure shows the effect of loading the quadword from both Big-Endian storage and Little-Endian storage.
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 123 illustrates both Big-Endian and Little-Endian storage images of array B.

*Big-Endian storage image of array B*

<table>
<thead>
<tr>
<th>0x0000:</th>
<th>01 23 45 67</th>
<th>00 11 22 33</th>
<th>44 55 66 77</th>
<th>88 99 AABB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0010:</td>
<td>CCDDEEFF</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Little-Endian storage image of array B*

<table>
<thead>
<tr>
<th>0x0000:</th>
<th>67 45 23 01</th>
<th>33 22 11 00</th>
<th>77 66 55 44</th>
<th>BBAA 99 88</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0010:</td>
<td>FF EEDDCO</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 123. Storage images of array B

Though this example shows the array starting at a quadword-aligned address, if the subject data of interest are elements 1 through 4, accessing elements 1 through 4 of array B involves an unaligned quadword storage access that spans two aligned quadwords.

### Loading an Unaligned Quadword from Big-Endian Storage

Loading elements from elements 1 through 4 of B (see Figure 123) 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*

<table>
<thead>
<tr>
<th>0x0000:</th>
<th>01 23 45 67</th>
<th>00 11 22 33</th>
<th>44 55 66 77</th>
<th>88 99 AABB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0010:</td>
<td>CCDDEEFF</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

# Assumptions

- GPR[Ra] = address of B
- GPR[Rb] = 4 (index to B[1])

```c
lxvw4x Xt,Ra,Rb
```

Xt: 00 11 22 33 44 55 66 77 88 99 AABB CCDDEEFF

Figure 124. 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 123) 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*

<table>
<thead>
<tr>
<th>0x0000:</th>
<th>67 45 23 01</th>
<th>33 22 11 00</th>
<th>77 66 55 44</th>
<th>BBAA 99 88</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0010:</td>
<td>FF EEDDCO</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

# Assumptions

- GPR[A] = address of B
- GPR[B] = 4 (index to B[1])

```c
lxvw4x Xt,Ra,Rb
```

Xt: 00 11 22 33 44 55 66 77 88 99 AABB CCDDEEFF

Figure 125. 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 123) into VR[VT] involves an unaligned quadword storage access.

VSX supports word-aligned vector and scalar storage accesses using Big-Endian byte ordering.

Storing an Unaligned Quadword to Little-Endian Storage

Storing a VSR to elements 1 through 4 of B (see Figure 123) into VR[VT] involves an unaligned quadword storage access.

VSX supports word-aligned vector and scalar storage accesses using Little-Endian byte ordering.

### 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

There are two basic forms of scalar load and scalar store instructions, word and doubleword. VSX Scalar Load instructions place a copy of the contents of the addressed word or doubleword in storage into the left-most word or doubleword element of the target VSR. The contents of the right-most element(s) of the target VSR are undefined. VSX Scalar Store instructions place a copy of the contents of the left-most word or doubleword element in the source VSR into the addressed word or doubleword in storage.

There are two basic forms of vector load and vector store instructions, a vector of 4 word elements and a vector of two doublewords. Both forms access a quadword in storage.

There is one basic form of vector load and splat instruction, doubleword. VSX Vector Load and Splat instruction places a copy of the contents of the addressed doubleword in storage into both doubleword elements of the target VSR.

7.6.1.1.1 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 Dword</td>
<td>480</td>
</tr>
<tr>
<td>lxsdx</td>
<td>Load VSX Scalar Dword Indexed</td>
<td>480</td>
</tr>
<tr>
<td>lxsibzx</td>
<td>Load VSX Scalar as Integer Byte &amp; Zero Indexed</td>
<td>482</td>
</tr>
<tr>
<td>lxsihax</td>
<td>Load VSX Scalar as Integer Hword &amp; Zero Indexed</td>
<td>482</td>
</tr>
<tr>
<td>lxsiwax</td>
<td>Load VSX Scalar as Integer Word Algebraic Indexed</td>
<td>483</td>
</tr>
<tr>
<td>lxsizwx</td>
<td>Load VSX Scalar as Integer Word &amp; Zero Indexed</td>
<td>484</td>
</tr>
<tr>
<td>lxssp</td>
<td>Load VSX Scalar Single-Precision</td>
<td>485</td>
</tr>
<tr>
<td>lxsspx</td>
<td>Load VSX Scalar Single-Precision Indexed</td>
<td>485</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>stxsd</td>
<td>Store VSX Scalar Dword</td>
<td>498</td>
</tr>
<tr>
<td>stxsdx</td>
<td>Store VSX Scalar Dword Indexed</td>
<td>498</td>
</tr>
<tr>
<td>stxsibx</td>
<td>Store VSX Scalar as Integer Byte Indexed</td>
<td>499</td>
</tr>
<tr>
<td>stxsihx</td>
<td>Store VSX Scalar as Integer Hword Indexed</td>
<td>499</td>
</tr>
<tr>
<td>stxsiwx</td>
<td>Store VSX Scalar as Integer Word Indexed</td>
<td>500</td>
</tr>
<tr>
<td>stxssp</td>
<td>Store VSX Scalar Single-Precision</td>
<td>501</td>
</tr>
<tr>
<td>stxsspx</td>
<td>Store VSX Scalar Single-Precision Indexed</td>
<td>502</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>492</td>
</tr>
<tr>
<td>lxvb16x</td>
<td>Load VSX Vector Byte*16 Indexed</td>
<td>487</td>
</tr>
<tr>
<td>lxvd2x</td>
<td>Load VSX Vector Dword*2 Indexed</td>
<td>488</td>
</tr>
<tr>
<td>lxvh8x</td>
<td>Load VSX Vector Hword*8 Indexed</td>
<td>495</td>
</tr>
<tr>
<td>lxvw4x</td>
<td>Load VSX Vector Word*4 Indexed</td>
<td>496</td>
</tr>
<tr>
<td>lxvx</td>
<td>Load VSX Vector Indexed</td>
<td>492</td>
</tr>
</tbody>
</table>

Table 10. VSX Vector Load Instructions
### 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>lxvdsx</td>
<td>Load VSX Vector Dword and Splat Indexed</td>
<td>494</td>
</tr>
<tr>
<td>lxvwsx</td>
<td>Load VSX Vector Word &amp; Splat Indexed</td>
<td>497</td>
</tr>
</tbody>
</table>

### Table 12. VSX Vector Load with Length 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>489</td>
</tr>
<tr>
<td>lxvll</td>
<td>Load VSX Vector with Length Left-justified</td>
<td>491</td>
</tr>
</tbody>
</table>

### Table 13. VSX Vector Store Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>stxv</td>
<td>Store VSX Vector</td>
<td>507</td>
</tr>
<tr>
<td>stxvb16x</td>
<td>Store VSX Vector Byte*16 Indexed</td>
<td>503</td>
</tr>
<tr>
<td>stxvd2x</td>
<td>Store VSX Vector Dword*2 Indexed</td>
<td>504</td>
</tr>
<tr>
<td>stxvh8x</td>
<td>Store VSX Vector Hword*8 Indexed</td>
<td>505</td>
</tr>
<tr>
<td>stxvw4x</td>
<td>Store VSX Vector Word*4 Indexed</td>
<td>506</td>
</tr>
<tr>
<td>stxvx</td>
<td>Store VSX Vector Indexed</td>
<td>510</td>
</tr>
</tbody>
</table>

### Table 14. VSX Vector Store w/ 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>507</td>
</tr>
<tr>
<td>stxvll</td>
<td>Store VSX Vector with Length Left-justified</td>
<td>509</td>
</tr>
</tbody>
</table>
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>xsabstdp</td>
<td>VSX Scalar Absolute Double-Precision</td>
<td>512</td>
</tr>
<tr>
<td>xsabsp</td>
<td>VSX Scalar Absolute Quad-Precision</td>
<td>512</td>
</tr>
<tr>
<td>xscpsgndp</td>
<td>VSX Scalar Copy Sign Double-Precision</td>
<td>533</td>
</tr>
<tr>
<td>xscpsgnqp</td>
<td>VSX Scalar Copy Sign Quad-Precision</td>
<td>533</td>
</tr>
<tr>
<td>xsnabsdp</td>
<td>VSX Scalar Negative Absolute Double-Precision</td>
<td>606</td>
</tr>
<tr>
<td>xsnabsqp</td>
<td>VSX Scalar Negative Absolute Quad-Precision</td>
<td>606</td>
</tr>
<tr>
<td>xsnegdp</td>
<td>VSX Scalar Negate Double-Precision</td>
<td>607</td>
</tr>
<tr>
<td>xsnegqp</td>
<td>VSX Scalar Negate Quad-Precision</td>
<td>607</td>
</tr>
</tbody>
</table>

Table 15: 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>xvabstdp</td>
<td>VSX Vector Absolute Value Double-Precision</td>
<td>658</td>
</tr>
<tr>
<td>xvabssp</td>
<td>VSX Vector Absolute Value Single-Precision</td>
<td>658</td>
</tr>
<tr>
<td>xvcpsgndp</td>
<td>VSX Vector Copy Sign Double-Precision</td>
<td>671</td>
</tr>
<tr>
<td>xvcpsgnsp</td>
<td>VSX Vector Copy Sign Single-Precision</td>
<td>671</td>
</tr>
<tr>
<td>xvnabsdp</td>
<td>VSX Vector Negative Absolute Value Double-Precision</td>
<td>725</td>
</tr>
<tr>
<td>xvnabsssp</td>
<td>VSX Vector Negative Absolute Value Single-Precision</td>
<td>725</td>
</tr>
<tr>
<td>xvnegdp</td>
<td>VSX Vector Negate Double-Precision</td>
<td>726</td>
</tr>
<tr>
<td>xvnegsp</td>
<td>VSX Vector Negate Single-Precision</td>
<td>726</td>
</tr>
</tbody>
</table>

Table 16: 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>xsmadddp</td>
<td>VSX Scalar Add Double-Precision</td>
<td>513</td>
</tr>
<tr>
<td>xsmaddqp[o]</td>
<td>VSX Scalar Add Quad-Precision [using round to Odd]</td>
<td>520</td>
</tr>
<tr>
<td>xsmaddsp</td>
<td>VSX Scalar Add Single-Precision</td>
<td>518</td>
</tr>
<tr>
<td>xsdivdp</td>
<td>VSX Scalar Divide Double-Precision</td>
<td>562</td>
</tr>
<tr>
<td>xsdivqp[o]</td>
<td>VSX Scalar Divide Quad-Precision [using round to Odd]</td>
<td>564</td>
</tr>
<tr>
<td>xsdivsp</td>
<td>VSX Scalar Divide Single-Precision</td>
<td>566</td>
</tr>
<tr>
<td>xsmulp</td>
<td>VSX Scalar Multiply Double-Precision</td>
<td>600</td>
</tr>
<tr>
<td>xsmulqp[o]</td>
<td>VSX Scalar Multiply Quad-Precision [using round to Odd]</td>
<td>602</td>
</tr>
<tr>
<td>xsmuls</td>
<td>VSX Scalar Multiply Single-Precision</td>
<td>604</td>
</tr>
<tr>
<td>xssqrtddp</td>
<td>VSX Scalar Square Root Double-Precision</td>
<td>641</td>
</tr>
<tr>
<td>xssqrtqdp</td>
<td>VSX Scalar Square Root Quad-Precision [using round to Odd]</td>
<td>642</td>
</tr>
<tr>
<td>xssqrtsp</td>
<td>VSX Scalar Square Root Single-Precision</td>
<td>644</td>
</tr>
<tr>
<td>xsusbdp</td>
<td>VSX Scalar Subtract Double-Precision</td>
<td>645</td>
</tr>
<tr>
<td>xsusbqp[o]</td>
<td>VSX Scalar Subtract Quad-Precision [using round to Odd]</td>
<td>647</td>
</tr>
<tr>
<td>xsusbsp</td>
<td>VSX Scalar Subtract Single-Precision</td>
<td>649</td>
</tr>
</tbody>
</table>

Table 17: VSX Scalar BFP Elementary Arithmetic Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>xsmaddaddp</td>
<td>VSX Scalar Multiply-Add Type-A Double-Precision</td>
<td>570</td>
</tr>
<tr>
<td>xsmaddasp</td>
<td>VSX Scalar Multiply-Add Type-A Single-Precision</td>
<td>573</td>
</tr>
</tbody>
</table>

Table 18: VSX Scalar BFP Multiply-Add-class Instructions
### Table 18. 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><code>xsmadddp</code></td>
<td>VSX Scalar Multiply-Add Type-M Double-Precision</td>
<td>570</td>
</tr>
<tr>
<td><code>xsmaddmsp</code></td>
<td>VSX Scalar Multiply-Add Type-M Single-Precision</td>
<td>573</td>
</tr>
<tr>
<td><code>xsmaddqpo</code></td>
<td>VSX Scalar Multiply-Add Quad-Precision [using round to Odd]</td>
<td>576</td>
</tr>
<tr>
<td><code>xsmsubdp</code></td>
<td>VSX Scalar Multiply-Subtract Type-A Double-Precision</td>
<td>591</td>
</tr>
<tr>
<td><code>xsmsubasp</code></td>
<td>VSX Scalar Multiply-Subtract Type-A Single-Precision</td>
<td>594</td>
</tr>
<tr>
<td><code>xsmsubmdp</code></td>
<td>VSX Scalar Multiply-Subtract Type-M Double-Precision</td>
<td>591</td>
</tr>
<tr>
<td><code>xsmsubmsp</code></td>
<td>VSX Scalar Multiply-Subtract Type-M Single-Precision</td>
<td>594</td>
</tr>
<tr>
<td><code>xsmsubqpo</code></td>
<td>VSX Scalar Multiply-Subtract Quad-Precision [using round to Odd]</td>
<td>597</td>
</tr>
<tr>
<td><code>xsnmaddadp</code></td>
<td>VSX Scalar Negative Multiply-Add Type-A Double-Precision</td>
<td>608</td>
</tr>
<tr>
<td><code>xsnmaddasp</code></td>
<td>VSX Scalar Negative Multiply-Add Type-A Single-Precision</td>
<td>613</td>
</tr>
<tr>
<td><code>xsnmaddmdp</code></td>
<td>VSX Scalar Negative Multiply-Add Type-M Double-Precision</td>
<td>608</td>
</tr>
<tr>
<td><code>xsnmaddmsp</code></td>
<td>VSX Scalar Negative Multiply-Add Type-M Single-Precision</td>
<td>613</td>
</tr>
<tr>
<td><code>xsnmaddqpo</code></td>
<td>VSX Scalar Negative Multiply-Add Quad-Precision [using round to Odd]</td>
<td>616</td>
</tr>
<tr>
<td><code>xsnmsubadp</code></td>
<td>VSX Scalar Negative Multiply-Subtract Type-A Double-Precision</td>
<td>619</td>
</tr>
<tr>
<td><code>xsnmsubasp</code></td>
<td>VSX Scalar Negative Multiply-Subtract Type-A Single-Precision</td>
<td>622</td>
</tr>
<tr>
<td><code>xsnmsubmdp</code></td>
<td>VSX Scalar Negative Multiply-Subtract Type-M Double-Precision</td>
<td>619</td>
</tr>
<tr>
<td><code>xsnmsubmsp</code></td>
<td>VSX Scalar Negative Multiply-Subtract Type-M Single-Precision</td>
<td>622</td>
</tr>
<tr>
<td><code>xsnmsubqpo</code></td>
<td>VSX Scalar Negative Multiply-Subtract Quad-Precision [using round to Odd]</td>
<td>625</td>
</tr>
</tbody>
</table>

### Table 19. VSX Scalar Software BFP Divide/Square Root Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>xrsredp</code></td>
<td>VSX Scalar Reciprocal Estimate Double-Precision</td>
<td>632</td>
</tr>
<tr>
<td><code>xrsresp</code></td>
<td>VSX Scalar Reciprocal Estimate Single-Precision</td>
<td>633</td>
</tr>
<tr>
<td><code>xrsqrtdp</code></td>
<td>VSX Scalar Reciprocal Square Root Estimate Double-Precision</td>
<td>639</td>
</tr>
<tr>
<td><code>xrsqfrtesp</code></td>
<td>VSX Scalar Reciprocal Square Root Estimate Single-Precision</td>
<td>640</td>
</tr>
<tr>
<td><code>xstdvdp</code></td>
<td>VSX Scalar Test for software Divide Double-Precision</td>
<td>651</td>
</tr>
<tr>
<td><code>xstsdvdp</code></td>
<td>VSX Scalar Test for software Square Root Double-Precision</td>
<td>652</td>
</tr>
</tbody>
</table>

### 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><code>xvadddp</code></td>
<td>VSX Vector Add Double-Precision</td>
<td>659</td>
</tr>
<tr>
<td><code>xvaddsp</code></td>
<td>VSX Vector Add Single-Precision</td>
<td>663</td>
</tr>
<tr>
<td><code>xvdvdp</code></td>
<td>VSX Vector Divide Double-Precision</td>
<td>696</td>
</tr>
<tr>
<td><code>xvdivsp</code></td>
<td>VSX Vector Divide Single-Precision</td>
<td>696</td>
</tr>
<tr>
<td><code>xvmulp</code></td>
<td>VSX Vector Multiply Double-Precision</td>
<td>721</td>
</tr>
<tr>
<td><code>xvmulsp</code></td>
<td>VSX Vector Multiply Single-Precision</td>
<td>723</td>
</tr>
<tr>
<td><code>xvsqtdp</code></td>
<td>VSX Vector Square Root Double-Precision</td>
<td>751</td>
</tr>
<tr>
<td><code>xvsqrtsp</code></td>
<td>VSX Vector Square Root Single-Precision</td>
<td>752</td>
</tr>
<tr>
<td><code>xvsubdp</code></td>
<td>VSX Vector Subtract Double-Precision</td>
<td>753</td>
</tr>
<tr>
<td><code>xvsubsp</code></td>
<td>VSX Vector Subtract Single-Precision</td>
<td>755</td>
</tr>
</tbody>
</table>

Table 20. VSX Vector BFP Elementary Arithmetic Instructions
### Table 21. 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>xvmaddadp</td>
<td>VSX Vector Multiply-Add Type-A Double-Precision</td>
<td>701</td>
</tr>
<tr>
<td>xvmaddasp</td>
<td>VSX Vector Multiply-Add Type-A Single-Precision</td>
<td>704</td>
</tr>
<tr>
<td>xvmaddmdp</td>
<td>VSX Vector Multiply-Add Type-M Double-Precision</td>
<td>701</td>
</tr>
<tr>
<td>xvmaddmsp</td>
<td>VSX Vector Multiply-Add Type-M Single-Precision</td>
<td>704</td>
</tr>
<tr>
<td>xvmsubadp</td>
<td>VSX Vector Multiply-Subtract Type-A Double-Precision</td>
<td>715</td>
</tr>
<tr>
<td>xvmsubasp</td>
<td>VSX Vector Multiply-Subtract Type-A Single-Precision</td>
<td>718</td>
</tr>
<tr>
<td>xvmsubmdp</td>
<td>VSX Vector Multiply-Subtract Type-M Double-Precision</td>
<td>715</td>
</tr>
<tr>
<td>xvmsubmsp</td>
<td>VSX Vector Multiply-Subtract Type-M Single-Precision</td>
<td>718</td>
</tr>
<tr>
<td>xvnmaddadp</td>
<td>VSX Vector Negative Multiply-Add Type-A Double-Precision</td>
<td>727</td>
</tr>
<tr>
<td>xvnmaddasp</td>
<td>VSX Vector Negative Multiply-Add Type-A Single-Precision</td>
<td>732</td>
</tr>
<tr>
<td>xvnmaddmdp</td>
<td>VSX Vector Negative Multiply-Add Type-M Double-Precision</td>
<td>727</td>
</tr>
<tr>
<td>xvnmaddmsp</td>
<td>VSX Vector Negative Multiply-Add Type-M Single-Precision</td>
<td>732</td>
</tr>
<tr>
<td>xvmmsubadp</td>
<td>VSX Vector Negative Multiply-Subtract Type-A Double-Precision</td>
<td>735</td>
</tr>
<tr>
<td>xvmmsubasp</td>
<td>VSX Vector Negative Multiply-Subtract Type-A Single-Precision</td>
<td>738</td>
</tr>
<tr>
<td>xvmmsubmdp</td>
<td>VSX Vector Negative Multiply-Subtract Type-M Double-Precision</td>
<td>735</td>
</tr>
<tr>
<td>xvmmsubmsp</td>
<td>VSX Vector Negative Multiply-Subtract Type-M Single-Precision</td>
<td>738</td>
</tr>
</tbody>
</table>

### Table 22. 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>744</td>
</tr>
<tr>
<td>xvresp</td>
<td>VSX Vector Reciprocal Estimate Single-Precision</td>
<td>745</td>
</tr>
<tr>
<td>xvrsqrtedp</td>
<td>VSX Vector Reciprocal Square Root Estimate Double-Precision</td>
<td>748</td>
</tr>
<tr>
<td>xvrsqrtesp</td>
<td>VSX Vector Reciprocal Square Root Estimate Single-Precision</td>
<td>750</td>
</tr>
<tr>
<td>xtddivdp</td>
<td>VSX Vector Test for software Divide Double-Precision</td>
<td>757</td>
</tr>
<tr>
<td>xtddivsp</td>
<td>VSX Vector Test for software Divide Single-Precision</td>
<td>758</td>
</tr>
<tr>
<td>xtsqrttdp</td>
<td>VSX Vector Test for software Square Root Double-Precision</td>
<td>759</td>
</tr>
<tr>
<td>xtsqrtsp</td>
<td>VSX Vector Test for software Square Root Single-Precision</td>
<td>759</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>
<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>527</td>
</tr>
<tr>
<td>xscmpoqp</td>
<td>VSX Scalar Compare Ordered Quad-Precision</td>
<td>529</td>
</tr>
<tr>
<td>xscmpudp</td>
<td>VSX Scalar Compare Unordered Double-Precision</td>
<td>530</td>
</tr>
<tr>
<td>xscmpuqp</td>
<td>VSX Scalar Compare Unordered Quad-Precision</td>
<td>532</td>
</tr>
</tbody>
</table>

Table 23. VSX Scalar BFP 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>524</td>
</tr>
<tr>
<td>xscmpgedp</td>
<td>VSX Scalar Compare Greater Than or Equal Double-Precision</td>
<td>525</td>
</tr>
<tr>
<td>xscmpgdp</td>
<td>VSX Scalar Compare Greater Than Double-Precision</td>
<td>526</td>
</tr>
</tbody>
</table>

Table 24. VSX Scalar BFP Predicate Compare 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>581</td>
</tr>
<tr>
<td>xsmaxdp</td>
<td>VSX Scalar Maximum Double-Precision</td>
<td>579</td>
</tr>
<tr>
<td>xsmaxjdp</td>
<td>VSX Scalar Maximum Type-J Double-Precision</td>
<td>583</td>
</tr>
<tr>
<td>xsmindp</td>
<td>VSX Scalar Minimum Type-C Double-Precision</td>
<td>587</td>
</tr>
<tr>
<td>xsmindjdp</td>
<td>VSX Scalar Minimum Type-J Double-Precision</td>
<td>589</td>
</tr>
</tbody>
</table>

Table 25. VSX Scalar BFP Maximum/Minimum Instructions

7.6.1.4.2 VSX Vector BFP 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>665</td>
</tr>
<tr>
<td>xvcmp eqsp.[]</td>
<td>VSX Vector Compare Equal To Single-Precision</td>
<td>666</td>
</tr>
<tr>
<td>xvcmpgedp.[]</td>
<td>VSX Vector Compare Greater Than or Equal To Double-Precision</td>
<td>667</td>
</tr>
<tr>
<td>xvcmpgesp.[]</td>
<td>VSX Vector Compare Greater Than or Equal To Single-Precision</td>
<td>668</td>
</tr>
<tr>
<td>xvcmpgt dp.[]</td>
<td>VSX Vector Compare Greater Than Double-Precision</td>
<td>669</td>
</tr>
<tr>
<td>xvcmpgt sp.[]</td>
<td>VSX Vector Compare Greater Than Single-Precision</td>
<td>670</td>
</tr>
</tbody>
</table>

Table 26. VSX Vector BFP Predicate Compare Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>xvmaxdp</td>
<td>VSX Vector Maximum Double-Precision</td>
<td>707</td>
</tr>
<tr>
<td>xvmaxsp</td>
<td>VSX Vector Maximum Single-Precision</td>
<td>709</td>
</tr>
<tr>
<td>xvmin dp</td>
<td>VSX Vector Minimum Double-Precision</td>
<td>711</td>
</tr>
<tr>
<td>xvmin sp</td>
<td>VSX Vector Minimum Single-Precision</td>
<td>713</td>
</tr>
</tbody>
</table>

Table 27. 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>636</td>
</tr>
<tr>
<td>xsrsp</td>
<td>VSX Scalar Round Double-Precision to Single-Precision</td>
<td>638</td>
</tr>
</tbody>
</table>

Table 28. 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 w/ round Double-Precision to Half-Precision format</td>
<td>534</td>
</tr>
<tr>
<td>xscvdpsp</td>
<td>VSX Scalar Convert w/ round Double-Precision to Single-Precision format</td>
<td>536</td>
</tr>
<tr>
<td>xscvdpspn</td>
<td>VSX Scalar Convert Double-Precision to Single-Precision format Non-signalling</td>
<td>537</td>
</tr>
<tr>
<td>xscvqdpdp[o]</td>
<td>VSX Scalar Convert w/ round Quad-Precision to Double-Precision format [using round to Odd]</td>
<td>638</td>
</tr>
</tbody>
</table>

Table 29. 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>xvcvdpsp</td>
<td>VSX Vector Convert w/ round Double-Precision to Single-Precision format</td>
<td>672</td>
</tr>
<tr>
<td>xvcvshph</td>
<td>VSX Vector Convert w/ round Single-Precision to Half-Precision format</td>
<td>683</td>
</tr>
</tbody>
</table>

Table 30. 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>xscvdqqp</td>
<td>VSX Scalar Convert Double-Precision to Quad-Precision format</td>
<td>535</td>
</tr>
<tr>
<td>xscvhdpd</td>
<td>VSX Scalar Convert Half-Precision to Double-Precision format</td>
<td>546</td>
</tr>
<tr>
<td>xscvsdpdp</td>
<td>VSX Scalar Convert Single-Precision to Double-Precision format</td>
<td>557</td>
</tr>
<tr>
<td>xscvsdpdn</td>
<td>VSX Scalar Convert Single-Precision to Double-Precision format Non-signalling</td>
<td>558</td>
</tr>
</tbody>
</table>

Table 31. 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>xvcvhsp</td>
<td>VSX Vector Convert Half-Precision to Single-Precision format</td>
<td>681</td>
</tr>
<tr>
<td>xvcvsdp</td>
<td>VSX Vector Convert Single-Precision to Double-Precision format</td>
<td>682</td>
</tr>
</tbody>
</table>

Table 32. 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>628</td>
</tr>
<tr>
<td>xsrdpic</td>
<td>VSX Scalar Round to Double-Precision Integer Exact using Current rounding mode</td>
<td>629</td>
</tr>
<tr>
<td>xsrdpim</td>
<td>VSX Scalar Round to Double-Precision Integer using round towards -Infinity</td>
<td>630</td>
</tr>
<tr>
<td>xsrdpip</td>
<td>VSX Scalar Round to Double-Precision Integer using round towards +Infinity</td>
<td>630</td>
</tr>
<tr>
<td>xsrdpiz</td>
<td>VSX Scalar Round to Double-Precision Integer using round towards Zero</td>
<td>631</td>
</tr>
<tr>
<td>xsrqpi</td>
<td>VSX Scalar Round to Quad-Precision Integer</td>
<td>634</td>
</tr>
<tr>
<td>xsrqpix</td>
<td>VSX Scalar Round Quad-Precision to Integral Exact</td>
<td>634</td>
</tr>
<tr>
<td>xvrdpi</td>
<td>VSX Vector Round to Double-Precision Integer using round to Nearest Away</td>
<td>741</td>
</tr>
<tr>
<td>xvrdpic</td>
<td>VSX Vector Round to Double-Precision Integer Exact using Current rounding mode</td>
<td>741</td>
</tr>
<tr>
<td>xvrdpim</td>
<td>VSX Vector Round to Double-Precision Integer using round towards -Infinity</td>
<td>742</td>
</tr>
<tr>
<td>xvrdpip</td>
<td>VSX Vector Round to Double-Precision Integer using round towards +Infinity</td>
<td>742</td>
</tr>
<tr>
<td>xvrdpiz</td>
<td>VSX Vector Round to Double-Precision Integer using round towards Zero</td>
<td>743</td>
</tr>
</tbody>
</table>

Table 33.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>xvrdpi</td>
<td>VSX Vector Round to Double-Precision Integer using round to Nearest Away</td>
<td>741</td>
</tr>
<tr>
<td>xvrdpic</td>
<td>VSX Vector Round to Double-Precision Integer Exact using Current rounding mode</td>
<td>741</td>
</tr>
<tr>
<td>xvrdpim</td>
<td>VSX Vector Round to Double-Precision Integer using round towards -Infinity</td>
<td>742</td>
</tr>
<tr>
<td>xvrdpip</td>
<td>VSX Vector Round to Double-Precision Integer using round towards +Infinity</td>
<td>742</td>
</tr>
<tr>
<td>xvrdpiz</td>
<td>VSX Vector Round to Double-Precision Integer using round towards Zero</td>
<td>743</td>
</tr>
<tr>
<td>xvrspi</td>
<td>VSX Vector Round to Single-Precision Integer using round to Nearest Away</td>
<td>746</td>
</tr>
<tr>
<td>xvrspic</td>
<td>VSX Vector Round to Single-Precision Integer Exact using Current rounding mode</td>
<td>746</td>
</tr>
<tr>
<td>xvrspim</td>
<td>VSX Vector Round to Single-Precision Integer using round towards -Infinity</td>
<td>747</td>
</tr>
<tr>
<td>xvrspip</td>
<td>VSX Vector Round to Single-Precision Integer using round towards +Infinity</td>
<td>747</td>
</tr>
<tr>
<td>xvrspiz</td>
<td>VSX Vector Round to Single-Precision Integer using round towards Zero</td>
<td>748</td>
</tr>
</tbody>
</table>

Table 34.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 w/ truncate Double-Precision to Signed Dword format</td>
<td>537</td>
</tr>
<tr>
<td>xscvdpsxws</td>
<td>VSX Scalar Convert w/ truncate Double-Precision to Signed Word format</td>
<td>540</td>
</tr>
<tr>
<td>xscvdpuxds</td>
<td>VSX Scalar Convert w/ truncate Double-Precision to Unsigned Dword format</td>
<td>542</td>
</tr>
<tr>
<td>xscvdpuxws</td>
<td>VSX Scalar Convert w/ truncate Double-Precision to Unsigned Word format</td>
<td>544</td>
</tr>
<tr>
<td>xscvqpsdz</td>
<td>VSX Scalar Convert w/ truncate Quad-Precision to Signed Dword format</td>
<td>548</td>
</tr>
<tr>
<td>xscvqpswz</td>
<td>VSX Scalar Convert w/ truncate Quad-Precision to Signed Word format</td>
<td>550</td>
</tr>
<tr>
<td>xscvqpwdz</td>
<td>VSX Scalar Convert w/ truncate Quad-Precision to Unsigned Dword format</td>
<td>552</td>
</tr>
<tr>
<td>xscvqpuwz</td>
<td>VSX Scalar Convert w/ truncate Quad-Precision to Unsigned Word format</td>
<td>554</td>
</tr>
</tbody>
</table>

Table 35.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>xcvdpxsdxs</td>
<td>VSX Vector Convert w/ truncate Double-Precision to Signed Dword format</td>
<td>673</td>
</tr>
<tr>
<td>xcvdpxsdxs</td>
<td>VSX Vector Convert w/ truncate Double-Precision to Signed Word format</td>
<td>675</td>
</tr>
<tr>
<td>xcvdpuxdsx</td>
<td>VSX Vector Convert w/ truncate Double-Precision to Unsigned Dword format</td>
<td>677</td>
</tr>
<tr>
<td>xcvdpuxdsx</td>
<td>VSX Vector Convert w/ truncate Double-Precision to Unsigned Word format</td>
<td>679</td>
</tr>
<tr>
<td>xcvspuxdxs</td>
<td>VSX Vector Convert w/ truncate Single-Precision to Signed Dword format</td>
<td>684</td>
</tr>
<tr>
<td>xcvspuxdxs</td>
<td>VSX Vector Convert w/ truncate Single-Precision to Signed Word format</td>
<td>686</td>
</tr>
<tr>
<td>xcvspuxdxs</td>
<td>VSX Vector Convert w/ truncate Single-Precision to Unsigned Dword format</td>
<td>688</td>
</tr>
<tr>
<td>xcvspuxdxs</td>
<td>VSX Vector Convert w/ truncate Single-Precision to Unsigned Word format</td>
<td>690</td>
</tr>
</tbody>
</table>

Table 36. 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>xscvsdpqp</td>
<td>VSX Scalar Convert Signed Dword to Quad-Precision format</td>
<td>556</td>
</tr>
<tr>
<td>xscvsxddd</td>
<td>VSX Scalar Convert w/ round Signed Dword to Double-Precision format</td>
<td>559</td>
</tr>
<tr>
<td>xscvsxddd</td>
<td>VSX Scalar Convert w/ round Signed Dword to Single-Precision format</td>
<td>559</td>
</tr>
<tr>
<td>xscvuddp</td>
<td>VSX Scalar Convert Unsigned Dword to Quad-Precision format</td>
<td>560</td>
</tr>
<tr>
<td>xscvuxddp</td>
<td>VSX Scalar Convert w/ round Unsigned Dword to Double-Precision format</td>
<td>561</td>
</tr>
<tr>
<td>xscvuxddp</td>
<td>VSX Scalar Convert w/ round Unsigned Dword to Single-Precision format</td>
<td>561</td>
</tr>
</tbody>
</table>

Table 37. 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>xcvvsxddd</td>
<td>VSX Vector Convert w/ round Signed Dword to Double-Precision format</td>
<td>692</td>
</tr>
<tr>
<td>xcvvsxddd</td>
<td>VSX Vector Convert Signed Word to Double-Precision format</td>
<td>693</td>
</tr>
<tr>
<td>xcvvuxddp</td>
<td>VSX Vector Convert w/ round Unsigned Dword to Double-Precision format</td>
<td>694</td>
</tr>
<tr>
<td>xcvvuxddp</td>
<td>VSX Vector Convert Unsigned Word to Double-Precision format</td>
<td>695</td>
</tr>
<tr>
<td>xcvvsxddd</td>
<td>VSX Vector Convert w/ round Signed Dword to Single-Precision format</td>
<td>692</td>
</tr>
<tr>
<td>xcvvsxddd</td>
<td>VSX Vector Convert Signed Word to Single-Precision format</td>
<td>693</td>
</tr>
<tr>
<td>xcvvuxddp</td>
<td>VSX Vector Convert w/ round Unsigned Dword to Single-Precision format</td>
<td>694</td>
</tr>
<tr>
<td>xcvvuxddp</td>
<td>VSX Vector Convert Unsigned Word to Single-Precision format</td>
<td>695</td>
</tr>
</tbody>
</table>

Table 38. 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>xscmpexpdp</td>
<td>VSX Scalar Compare Exponents Double-Precision</td>
<td>522</td>
</tr>
<tr>
<td>xscmpexpqdp</td>
<td>VSX Scalar Compare Exponents Quad-Precision</td>
<td>523</td>
</tr>
<tr>
<td>xsieexpdp</td>
<td>VSX Scalar Insert Exponent Double-Precision</td>
<td>568</td>
</tr>
<tr>
<td>xsieexpqdp</td>
<td>VSX Scalar Insert Exponent Quad-Precision</td>
<td>569</td>
</tr>
<tr>
<td>xststdcddp</td>
<td>VSX Scalar Test Data Class Double-Precision</td>
<td>653</td>
</tr>
<tr>
<td>xststdcqd</td>
<td>VSX Scalar Test Data Class Quad-Precision</td>
<td>654</td>
</tr>
<tr>
<td>xststdcsp</td>
<td>VSX Scalar Test Data Class Single-Precision</td>
<td>655</td>
</tr>
<tr>
<td>xseexpdp</td>
<td>VSX Scalar Extract Exponent Double-Precision</td>
<td>656</td>
</tr>
<tr>
<td>xseexpqdp</td>
<td>VSX Scalar Extract Exponent Quad-Precision</td>
<td>656</td>
</tr>
</tbody>
</table>

Table 39. VSX Scalar BFP Math Support Instructions
7.6.1.11.2 VSX Vector BFP Math Support Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>xsxsigdp</td>
<td>VSX Scalar Extract Significand Double-Precision</td>
<td>657</td>
</tr>
<tr>
<td>xsxsigqp</td>
<td>VSX Scalar Extract Significand Quad-Precision</td>
<td>657</td>
</tr>
</tbody>
</table>

Table 39. VSX Scalar BFP Math Support Instructions

7.6.1.12 VSX Vector Logical Instructions

7.6.1.12.1 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>767</td>
</tr>
<tr>
<td>xxlandc</td>
<td>VSX Vector Logical AND with Complement</td>
<td>767</td>
</tr>
<tr>
<td>xxleqv</td>
<td>VSX Vector Logical Equivalence</td>
<td>768</td>
</tr>
<tr>
<td>xxlnand</td>
<td>VSX Vector Logical NAND</td>
<td>768</td>
</tr>
<tr>
<td>xxlnor</td>
<td>VSX Vector Logical NOR</td>
<td>769</td>
</tr>
<tr>
<td>xxlor</td>
<td>VSX Vector Logical OR</td>
<td>770</td>
</tr>
<tr>
<td>xxlxor</td>
<td>VSX Vector Logical XOR</td>
<td>770</td>
</tr>
</tbody>
</table>

Table 41. VSX Logical Instructions

7.6.1.12.2 VSX Vector Select Instruction

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>xxsel</td>
<td>VSX Vector Select</td>
<td>773</td>
</tr>
</tbody>
</table>

Table 42. VSX Vector Select Instruction

7.6.1.13 VSX Vector Permute-class Instructions

7.6.1.13.1 VSX Vector Byte-Reverse Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>xxbrd</td>
<td>VSX Vector Byte-Reverse Dword</td>
<td>764</td>
</tr>
<tr>
<td>xxbrh</td>
<td>VSX Vector Byte-Reverse Hword</td>
<td>764</td>
</tr>
<tr>
<td>xxbrq</td>
<td>VSX Vector Byte-Reverse Qword</td>
<td>765</td>
</tr>
<tr>
<td>xxbrw</td>
<td>VSX Vector Byte-Reverse Word</td>
<td>765</td>
</tr>
</tbody>
</table>

Table 43. VSX Vector Byte-Reverse Instructions
### 7.6.1.13.2 VSX Vector Insert/Extract Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>xxextractuw</td>
<td>VSX Vector Extract Unsigned Word</td>
<td>766</td>
</tr>
<tr>
<td>xxinsertw</td>
<td>VSX Vector Insert Word</td>
<td>766</td>
</tr>
</tbody>
</table>

Table 44. VSX Vector Insert/Extract Instructions

### 7.6.1.13.3 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>771</td>
</tr>
<tr>
<td>xxmrglw</td>
<td>VSX Vector Merge Low Word</td>
<td>771</td>
</tr>
</tbody>
</table>

Table 45. VSX Vector Merge Instructions

### 7.6.1.13.4 VSX Vector Splat Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>xxsplitib</td>
<td>VSX Vector Splat Immediate Byte</td>
<td>774</td>
</tr>
<tr>
<td>xxsplitw</td>
<td>VSX Vector Splat Word</td>
<td>774</td>
</tr>
</tbody>
</table>

Table 46. VSX Vector Splat Instructions

### 7.6.1.13.5 VSX Vector Permute Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>xxpermidi</td>
<td>VSX Vector Permute Dword Immediate</td>
<td>773</td>
</tr>
<tr>
<td>xxperm</td>
<td>VSX Vector Permute</td>
<td>772</td>
</tr>
<tr>
<td>xxpermr</td>
<td>VSX Vector Permute Right-indexed</td>
<td>772</td>
</tr>
</tbody>
</table>

Table 47. VSX Vector Permute Instruction

### 7.6.1.13.6 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>774</td>
</tr>
</tbody>
</table>

Table 48. VSX Vector Shift Left Double 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 \( x \).

\( 1x \)
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

**AddDP(x,y)**

- *x* and *y* are double-precision floating-point values.
  - 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 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.

**AddSP(x,y)**

- *x* and *y* are single-precision floating-point values.
  - 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 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* added to *y*, having unbounded range and precision.

**bfp_ABSOLUTE(x)**

- *x* is a binary floating-point value represented in the working floating-point format.
  - Return *x* with sign set to 0.

**bfp_ADD(x, y)**

- *x* is a binary floating-point value represented in the working floating-point format.
  - *y* is a binary floating-point value represented in the working floating-point 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 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 working floating-point format.
  - *y* is a binary floating-point value represented in the working floating-point format.
  - Return 0b0 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 working floating-point format.
  y is a binary floating-point value represented in the working floating-point 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 working floating-point format.
  y is a binary floating-point value represented in the working floating-point 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_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.

- \( \text{result.class.Denormal} \) is set to 1.
- \( \text{result.sign} \) is set to the contents of bit 0 of \( x \).
- \( \text{result.exp} \) is set to the value -14.

- The contents of bit 0 of \( \text{result.significand} \) are set to 0.
- The contents of bits 1:10 of \( \text{result.significand} \) are set to the value of \( \text{fraction} \).

- \( \text{result.significand} \) is shifted left until the contents bit 0 of \( \text{result.significand} \) are equal to 1.
- \( \text{result.exponent} \) is decremented by the the number of bits \( \text{result.significand} \) was shifted.

Otherwise, do the following.

- \( \text{result.class.Normal} \) is set to 1.
- \( \text{result.sign} \) is set to the contents of bit 0 of \( x \).
- \( \text{result.exp} \) is set to the value of \( \text{exponent} \) subtracted by 15.

- The contents of bit 0 of \( \text{result.significand} \) are set to 1.
- The contents of bits 1:10 of \( \text{result.significand} \) are set to the value of \( \text{fraction} \).

Return \( \text{result} \).
bfp_CONVERT_FROM_BFP32(x)
    x is a floating-point value represented in single-precision format.

    Let exponent be the contents of bits 1:8 of x.
    Let fraction be the contents of bits 9:31 of x.

    Let result.sign be initialized to 0.
    Let result.exponent be initialized to 0.
    Let result.significand be initialized to 0.
    Let result.class.SNaN be initialized to 0.
    Let result.class.QNaN be initialized to 0.
    Let result.class.Infinity be initialized to 0.
    Let result.class.Zero be initialized to 0.
    Let result.class.Denormal be initialized to 0.
    Let result.class.Normal be 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 1:23 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 1:23 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.exponent is set to the value -126.
        The contents of bit 0 of 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 working floating-point format).
bfp\_CONVERT\_FROM\_BFP128(x)

\( x \) is a binary floating-point value represented in quad-precision format.

Let \( \text{exponent} \) be the contents of bits 1:15 of \( x \).
Let \( \text{fraction} \) be the contents of bits 16:127 of \( x \).

\( \text{result}.\text{sign} \) is initialized to 0.
\( \text{result}.\text{exponent} \) is initialized to 0.
\( \text{result}.\text{significand} \) is initialized to 0.
\( \text{result}.\text{class}.\text{SNaN} \) is initialized to 0.
\( \text{result}.\text{class}.\text{QNaN} \) is initialized to 0.
\( \text{result}.\text{class}.\text{Infinity} \) is initialized to 0.
\( \text{result}.\text{class}.\text{Zero} \) is initialized to 0.
\( \text{result}.\text{class}.\text{Denormal} \) is initialized to 0.
\( \text{result}.\text{class}.\text{Normal} \) is initialized to 0.

If \( x \) is a \text{SNaN}, do the following.
\( \text{result}.\text{class}.\text{SNaN} \) is set to 1.
\( \text{result}.\text{sign} \) is set to the contents of bit 0 of \( x \).
The contents of bit 0 of \( \text{result}.\text{significand} \) are set to 0.
The contents of bits 1:112 of \( \text{result}.\text{significand} \) are set to the value of \( \text{fraction} \).
The contents of the rest of \( \text{result}.\text{significand} \) are set to 0.

Otherwise, if \( x \) is a \text{QNaN}, do the following.
\( \text{result}.\text{class}.\text{QNaN} \) is set to 1.
\( \text{result}.\text{sign} \) is set to the contents of bit 0 of \( x \).
The contents of bit 0 of \( \text{result}.\text{significand} \) are set to 0.
The contents of bits 1:112 of \( \text{result}.\text{significand} \) are set to the value of \( \text{fraction} \).
The contents of the rest of \( \text{result}.\text{significand} \) are set to 0.

Otherwise, if \( x \) is an \text{Infinity}, do the following.
\( \text{result}.\text{class}.\text{Infinity} \) is set to 1.
\( \text{result}.\text{sign} \) is set to the contents of bit 0 of \( x \).

Otherwise, if \( x \) is a \text{Zero}, do the following.
\( \text{result}.\text{class}.\text{Zero} \) is set to 1.
\( \text{result}.\text{sign} \) is set to the contents of bit 0 of \( x \).

Otherwise, if \( x \) is a \text{Denormal}, do the following.
\( \text{result}.\text{class}.\text{Denormal} \) is set to 1.
\( \text{result}.\text{sign} \) is set to the contents of bit 0 of \( x \).
\( \text{result}.\text{exp} \) is set to the value \(-16382\).
The contents of bit 0 of \( \text{result}.\text{significand} \) are set to 0.
The contents of bits 1:112 of \( \text{result}.\text{significand} \) are set to the value of \( \text{fraction} \).
The contents of the rest of \( \text{result}.\text{significand} \) are set to 0.
\( \text{result}.\text{significand} \) is shifted left until the contents bit 0 of \( \text{result}.\text{significand} \) are equal to 1.
\( \text{result}.\text{exponent} \) is decremented by the the number of bits \( \text{result}.\text{significand} \) was shifted.

Otherwise, do the following.
\( \text{result}.\text{class}.\text{Normal} \) is set to 1.
\( \text{result}.\text{sign} \) is set to the contents of bit 0 of \( x \).
\( \text{result}.\text{exp} \) is set to the value of \( \text{exponent} \) subtracted by \( 16383 \).
The contents of bit 0 of \( \text{result}.\text{significand} \) are set to 1.
The contents of bits 1:112 of \( \text{result}.\text{significand} \) are set to the value of \( \text{fraction} \).
The contents of the rest of \( \text{result}.\text{significand} \) are set to 0.

Return \( \text{result} \) (i.e., the value \( x \) in the working floating-point format).
bfp_CONVERT_FROM_SI64(x)

- \( x \) is an integer value represented in signed doubleword integer format.

\[
\begin{align*}
\text{result.sign} & \text{ is initialized to 0.} \\
\text{result.exponent} & \text{ is initialized to 0.} \\
\text{result.significand} & \text{ is initialized to 0.} \\
\text{result.class.SNaN} & \text{ is initialized to 0.} \\
\text{result.class.QNaN} & \text{ is initialized to 0.} \\
\text{result.class.Infinity} & \text{ is initialized to 0.} \\
\text{result.class.Zero} & \text{ is initialized to 0.} \\
\text{result.class.Denormal} & \text{ is initialized to 0.} \\
\text{result.class.Normal} & \text{ is initialized to 0.}
\end{align*}
\]

If \( x \) is equal to \( 0x0000_0000_0000_0000 \),
\[
\text{result.class.Zero} \text{ is set to 1.}
\]

Otherwise, do the following.
\[
\begin{align*}
\text{result.class.Normal} & \text{ is set to 1.} \\
\text{result.sign} & \text{ is set to the contents of bit 0 of } x. \\
\text{result.exponent} & \text{ is set to the value 64.} \\
\text{Bits }0:64\text{ of result.significand are set to the value of } x\text{ sign-extended to 65 bits.}
\end{align*}
\]

If bit 0 of result.significand is equal to 1,
\[
\begin{align*}
\text{result.sign} & \text{ is set to 1, and} \\
\text{result.significand} & \text{ is set to the value of the two’s complement of result.significand.}
\end{align*}
\]

If bit 0 of result.significand is equal to 0,
\[
\begin{align*}
\text{result.significand} & \text{ is shifted left until bit 0 of result.significand is equal to 1, and} \\
\text{result.exponent} & \text{ is decremented by the number of bits result.significand is shifted.}
\end{align*}
\]

Return result (i.e., the value \( x \) in the working floating-point format).
bfp_CONVERT_FROM_UI64(x)

\( x \) is an integer value represented in unsigned doubleword integer format.

Return \( x \) in the working floating-point 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 working floating-point format).

bfp_CONVERT_TO_BFP16(x)

\( x \) is a floating-point value represented in the 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.

**bfp_CONVERT_TO_BFP32(x)**

x is a floating-point value represented in the 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 result is set to the value of x.sign.
- Bits 1:9 of result are set to the value 0b1111_1111.
- Bits 9:31 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 1:31 of result are set to 0.

Otherwise, if x.exponent is less than -126 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, -126 - x.exponent.
- Bits 1:8 of result are set to 0b0000_0000.
- Bits 9:31 of result are set to bits 1:23 of x.significand shifted right by sh_cnt bits.

Otherwise, if x.exponent is less than -126 and UE=1, result is undefined.

Otherwise, if x.exponent is greater than 127 and OE=1, result is undefined.

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 9:31 of result are set to bits 1:23 of x.significand.

Return result.
bfp_CONVERT_TO_BFP64(x)

x is a floating-point value represented in the 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.class.Infinity=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 0.

Otherwise, if x.class.Zero=1, do the following.
  Bit 0 of result is set to the value of x.sign.
  Bits 1:63 of result are set to 0.

Otherwise, if x.exponent is less than -1022 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, -1022 - x.exponent.
  Bits 1:11 of result are set to 0b000_0000_0000.
  Bits 12:63 of result are set to bits 1:52 of x.significand shifted right by sh_cnt bits.

Otherwise, if x.exponent is less than -1022 and UE=1, result is undefined.
Otherwise, if x.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.sign.
  Bits 1:11 of result are set to the sum, x.exponent + 1023.
  Bits 12:63 of result are set to bits 1:52 of x.significand.

Return result.
**bfp_CONVERT_TO_BFP128(x)**

x is a quad-precision floating-point value that is represented in the working floating-point 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,
  - 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,
  - the contents of bits 1:15 of result are set to the value 0b000_0000_0000_0000,
  - the contents of bits 16:127 of result are set to the value 0x0000_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,
  - the contents of bits 1:15 of result are set to the value 0b111_1111_1111_1111,
  - the contents of bits 16:127 of result are set to the value 0x0000_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,
    - 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,
    - 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).

**bfp_CONVERT_TO_SI64(x)**

x is an integer value represented in the working floating-point format.

Return the value x in signed doubleword integer format.

**bfp_CONVERT_TO_UI64(x)**

x is an integer value represented in the working floating-point format.

Return the value x in 64-bit unsigned integer format.

**bfp_DENORM(x, y)**

x is an integer value specifying the target format’s Emin value.

- y is a binary floating-point value that is represented in the working floating-point format.

- If y.exponent is less than Emin, let sh_cnt be the value Emin - y.exponent.
- Otherwise, let sh_cnt be the value 0.

- y.significand, having unbounded precision, is shifted right by sh_cnt bits.
- y.exponent is incremented by sh_cnt.

Return y in the working floating-point format.
bfp_DIVIDE(x, y)

x is a binary floating-point value that is represented in the working floating-point format.
y is a binary floating-point value that is represented in the working floating-point format.

If x or y is an SNaN, vxsnan_flag is set to 1.
Otherwise, if x and y are infinities, vxidi_flag is set to 1.
Otherwise, if x and y are zeros, vxzdz_flag is set to 1.
Otherwise, if x is a finite value and y is a zero, 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 ÷ y, having unbounded range and precision.

bfp_INFINITY()
Return a positive floating-point infinity value, represented in the working format.

bfp_INITIALIZE(result)
result.class.Infinity ← 1
return(result)

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.class.Normal be set to 0.

Return x.

bfp_MULTIPLY(x, y)

x is a binary floating-point value represented in the working floating-point format.
y is a binary floating-point value represented in the working floating-point format.

If x or y is an SNaN, vxsnan_flag is set to 1.
Otherwise, if x is an infinity and y is a zero, vximz_flag is set to 1.
Otherwise, if x is a zero and y is an infinity, vximz_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 × y, having unbounded range and precision.
Chapter 7. Vector-Scalar Floating-Point Operations

\[ \text{bfp\_MULTIPLY\_ADD}(x, y, z) \]

- If \( x, y, \) or \( z \) is an SNaN, \( vxsnan\_flag \) is set to 1.
- Otherwise, if \( x \) is an infinity and \( y \) is a zero, \( vxiinz\_flag \) is set to 1.
- Otherwise, if \( x \) is a zero and \( y \) is an infinity, \( vxiinz\_flag \) is set to 1.
- Otherwise, if \( z \) and the product of \( x \times y \) are infinity values having opposite signs, \( vxisi\_flag \) 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 \( z \) is a QNaN, return \( z \).
- Otherwise, if \( z \) is an SNaN, return \( z \) 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, 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.

\[ \text{bfp\_NEGATE}(x) \]

- \( x \) is a binary floating-point value that is represented in the working floating-point format.
- Return \( x \) with its sign complemented.

\[ \text{bfp\_NMAX\_BFP16()} \]

- Return the largest, positive, normalized half-precision floating-point value, \((2 \cdot 2^{-10}) \times 2^{+15}\), represented in the working format.

\[ \text{bfp\_NMAX\_BFP64} \]

- Return the largest finite double-precision value (i.e., \( 2^{1024} \cdot 2^{1024-53} \)) in the working floating-point format.

\[ \text{bfp\_NMAX\_BFP80} \]

- Return the largest finite double-extended-precision value (i.e., \( 2^{16384} \cdot 2^{16384-65} \)) in the working floating-point format.

\[ \text{bfp\_NMAX\_BFP128} \]

- Return the largest finite quad-precision value (i.e., \( 2^{16384} \cdot 2^{16384-113} \)) in the working floating-point format.

\[ \text{bfp\_NMIN\_BFP16()} \]

- Return the smallest, positive, normalized half-precision floating-point value, \( 2^{-14} \), represented in the working format.
```python
return(result)

bfp_NMIN_BFP64
Return the smallest, positive, normalized double-precision 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 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 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 working floating-point 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 working floating-point 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_BFP16(x, y)

y is a normalized floating-point value represented in the 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 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 y.class.Zero | y.class.Infinity then return(y)

if y.class.QNaN | y.class.SNaN then do
  result ← y
  result.significand.bit[1] ← 1
  result.significand.bit[11:inf] ← 0
  result.class.SNaN ← 0
  result.class.QNaN ← 1
  vxsnan_flag ← y.class.SNaN
  return(result)
end

if bfp_COMPARE_LT(y,bfp_NMIN_BFP16()) then do
  if FPSCR.UE=0 then do
    do while y.exponent < -14    // denormalize y
      y.significand ← y.significand >> 1
      y.exponent ← y.exponent + 1
    end
  if x=0b00 then result ← bfp_ROUND_TO_BFP16_NEAR_EVEN(y)
  if x=0b01 then result ← bfp_ROUND_TO_BFP16_TRUNC(y)
  if x=0b10 then result ← bfp_ROUND_TO_BFP16_CEIL(y)
  if x=0b11 then result ← bfp_ROUND_TO_BFP16_FLOOR(y)
  do while result.significand.bit[0] = 0    // 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 ← bfp_ROUND_TO_BFP16_NEAR_EVEN(y)
if x=0b01 then result ← bfp_ROUND_TO_BFP16_TRUNC(y)
if x=0b10 then result ← bfp_ROUND_TO_BFP16_CEIL(y)
if x=0b11 then result ← bfp_ROUND_TO_BFP16_FLOOR(y)
if bfp_COMPARE_GT(result, bfp_NMAX_BFP16()) then do
    if OE=0 then do
        if x=0b00 then result ← sign ? bfp_NEGATE(bfp_INFINITY()) : bfp_INFINITY()
        if x=0b01 then result ← sign ? bfp_NEGATE(bfp_NMAX_BFP16()) : bfp_NMAX_BFP16()
        if x=0b10 then result ← sign ? bfp_NEGATE(bfp_NMAX_BFP16()) : bfp_INFINITY()
        if x=0b11 then result ← sign ? bfp_NEGATE(bfp_INFINITY()) : bfp_NMAX_BFP16()
        ox_flag ← 0b1
        xx_flag ← 0b1
        inc_flag ← 0bU
        return(result)
    end
    else do
        result.exponent ← result.exponent - 24
        ox_flag ← 1
    end
end
return(result)

bfp_ROUND_TO_BFP16_CEIL(x)
x is a normalized floating-point value represented in the 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 \( x \), \( \text{inc\_flag} \) is set to 1.

If the value returned is not equal to \( x \), \( \text{xx\_flag} \) is set to 1.

bfp_ROUND_TO_BFP16_FLOOR(x)
x is a normalized floating-point value represented in the 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 \( x \), \( \text{inc\_flag} \) is set to 1.

If the value returned is not equal to \( x \), \( \text{xx\_flag} \) is set to 1.

bfp_ROUND_TO_BFP16_NEAR_EVEN(x)
x is a normalized floating-point value represented in the 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 \( x \), \( \text{inc\_flag} \) is set to 1.

If the value returned is not equal to \( x \), \( \text{xx\_flag} \) is set to 1.
**bfp_ROUND_TO_BFP16_TRUNC(x)**

x is a normalized floating-point value represented in the 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-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.

**bfp_ROUND_TO_INTEGER(rmode, x)**

x is a binary floating-point value that is represented in the working floating-point format and has unbounded exponent range and significand precision.

If x is an SNaN, \( vxsnan \_flag \) 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 an Infinity, return x.

Otherwise, do the following.

- If \( rmode = 0b000 \) (Round to Nearest Even), 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 with the least-significant bit equal to 0 is used).
- If \( rmode = 0b001 \) (Round to Zero), 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 +Infinity), return the smallest double-precision floating-point integer value that is greater or equal in value to x.
- If \( rmode = 0b011 \) (Round towards -Infinity), 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 working floating-point 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 \( p \cdot 1 \) of the significand set to 1 if any of the bits to the right of bit \( p \cdot 1 \) of the significand of x are equal to 1, and all bits to the right of bit \( p \cdot 1 \) of the significand of the value returned are set to 0. Otherwise return x with all bits to the right of bit \( p \cdot 1 \) 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 working floating-point 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 working floating-point 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_ROUND_TO_BFP128(ro, rmode, x)

x is a normalized binary floating-point value that is represented in the working floating-point 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 quad-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_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
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 ← x.sign ? bfp_INFINITY : bfp_INFINITY
    if ro=0 & rmode=0b01 then r ← x.sign ? bfp_NMAX_BFP128 : bfp_NMAX_BFP128
    if ro=0 & rmode=0b10 then r ← x.sign ? bfp_NMAX_BFP128 : bfp_NMAX_BFP128
    if ro=0 & rmode=0b11 then r ← x.sign ? bfp_INFINITY : bfp_NMAX_BFP128
    if ro=1              then r ← x.sign ? bfp_NMAX_BFP128 : bfp_NMAX_BFP128
    r.sign ← x.sign
    ox_flag ← ob1
    xx_flag ← ob1
    inc_flag ← ob0
    return(r)
  end
  else do
    r.exponent ← r.exponent - 24576
    ox_flag ← 1
  end
end
return(r)
bfp_ROUND_TO_BFP80(rmode, x)

x is a normalized binary floating-point value that is represented in the working floating-point 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 +Infinity
rmode=0b11 Round towards -Infinity

Return the value x rounded to double-extended-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_BFP80 then do
  if FPSCR.UE=0 then do
    x ← bfp_DENORM(-16382, x)
    if rmode=0b00 then r ← bfp_ROUND_NEAR_EVEN(64, x)
    if rmode=0b01 then r ← bfp_ROUND_TRUNC(64, x)
    if rmode=0b10 then r ← bfp_ROUND_CEIL(64, x)
    if rmode=0b11 then r ← bfp_ROUND_FLOOR(64, x)
    ux_flag ← xx_flag
    return(r)
  end
else do
  x.exponent ← x.exponent + 24576
  ux_flag ← 1
end
end
if rmode=0b00 then r ← bfp_ROUND_NEAR_EVEN(64, x)
if rmode=0b01 then r ← bfp_ROUND_TRUNC(64, x)
if rmode=0b10 then r ← bfp_ROUND_CEIL(64, x)
if rmode=0b11 then r ← bfp_ROUND_FLOOR(64, x)
if bfp_ABSOLUTE(r)>bfp_NMAX_BFP80 then do
  if FPSCR.OE=0 then do
    if rmode=0b00 then r ← x.sign ? bfp_INFINITY : bfp_INFINITY
    if rmode=0b01 then r ← x.sign ? bfp_NMAX_BFP80 : bfp_NMAX_BFP80
    if rmode=0b10 then r ← x.sign ? bfp_NMAX_BFP80 : bfp_INFINITY
    if rmode=0b11 then r ← x.sign ? bfp_INFINITY : bfp_NMAX_BFP80
    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)
bfp_ROUND_TO_BFP64(ro, rmode, x)

x is a normalized binary floating-point value that is represented in the working floating-point 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   rmode=0     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
end
if bfp_ABSOLUTE(x)>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
    r.exponent  r.exponent - 1536
    ox_flag  1
  end
end
return(r)
\textbf{bfp\_SQUARE\_ROOT}(x)  
\hspace*{1em}x is a binary floating-point value that is represented in the working floating-point format and has unbounded exponent range and significand precision.

If \( x \) is an SNaN, \( \text{vxsnan\_flag} \) is set to 1.
Otherwise, if \( x \) is negative and non-zero, \( \text{vxsqrt\_flag} \) 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 negative, return the standard QNaN.
Otherwise, return the normalized square root of \( x \), having unbounded range and precision.

\textbf{ClassDP}(x,y)  
\hspace*{1em}Return a 5-bit characterization of the double-precision floating-point number \( x \).

\begin{align*}
0b10001 &= \text{Quiet NaN} \\
0b10011 &= -\text{Infinity} \\
0b10100 &= -\text{Normalized Number} \\
0b11000 &= -\text{Denormalized Number} \\
0b10010 &= -\text{Zero} \\
0b00010 &= +\text{Zero} \\
0b10100 &= +\text{Denormalized Number} \\
0b00100 &= +\text{Normalized Number} \\
0b00101 &= +\text{Infinity}
\end{align*}

\textbf{ClassSP}(x,y)  
\hspace*{1em}Return a 5-bit characterization of the single-precision floating-point number \( x \).

\begin{align*}
0b10001 &= \text{Quiet NaN} \\
0b10011 &= -\text{Infinity} \\
0b10100 &= -\text{Normalized Number} \\
0b11000 &= -\text{Denormalized Number} \\
0b10010 &= -\text{Zero} \\
0b00010 &= +\text{Zero} \\
0b10100 &= +\text{Denormalized Number} \\
0b00100 &= +\text{Normalized Number} \\
0b00101 &= +\text{Infinity}
\end{align*}

\textbf{CompareEQDP}(x,y)  
\hspace*{1em}x and y are double-precision floating-point values.

\begin{align*}
\text{If } x \text{ or } y \text{ is a NaN, return 0.} \\
\text{Otherwise, if } x \text{ is equal to } y, \text{ return 1.} \\
\text{Otherwise, return 0.}
\end{align*}

\textbf{CompareEQSP}(x,y)  
\hspace*{1em}x and y are single-precision floating-point values.

\begin{align*}
\text{If } x \text{ or } y \text{ is a NaN, return 0.} \\
\text{Otherwise, if } x \text{ is equal to } y, \text{ return 1.} \\
\text{Otherwise, return 0.}
\end{align*}

\textbf{CompareGTDP}(x,y)  
\hspace*{1em}x and y are double-precision floating-point values.

\begin{align*}
\text{If } x \text{ or } y \text{ is a NaN, return 0.} \\
\text{Otherwise, if } x \text{ is greater than } y, \text{ return 1.} \\
\text{Otherwise, return 0.}
\end{align*}
Chapter 7. Vector-Scalar Floating-Point Operations

CompareGTSP(x, y)
x and y are single-precision floating-point values.

If x or y is a NaN, return 0.
Otherwise, if x is greater than y, return 1.
Otherwise, return 0.

CompareLTDP(x, y)
x and y are double-precision floating-point values.

If x or y is a NaN, return 0.
Otherwise, if x is less than y, return 1.
Otherwise, return 0.

CompareLTSP(x, y)
x and y are single-precision floating-point values.

If x or y is a NaN, return 0.
Otherwise, if x is less than y, return 1.
Otherwise, return 0.

ConvertDPtoSD(x)
x is a floating-point value in double-precision format.

If x is a NaN,
  vxcki_flag is set to 1,
  vxsnan_flag is set to 1 if x is a SNaN, and
  return 0x8000_0000_0000_0000,
Otherwise, do the following.
  Let \( r \) be the value \( x \) truncated to an integral value.

  If \( r \) is greater than \( 2^{53} \cdot 2^{53} \),
    vxcki_flag is set to 1,
    return 0x7FFF_FFFF_FFFF_FFFF.

  Otherwise, if \( r \) is less than \( -2^{53} \),
    vxcki_flag is set to 1,
    return 0x8000_0000_0000_0000.

  Otherwise,
    xxl_flag is set to 1 if \( r \) is inexact.
    return \( r \) in 64-bit signed integer format.

ConvertDPtoSP(x)
x is a floating-point value in double-precision format.

If x is a NaN, vxsnan_flag is set to 1.
If x is a SNaN, returns x, converted to a QNaN, in single-precision floating-point format.
Otherwise, if x is a QNaN, an Infinity, or a Zero, returns x in single-precision floating-point format.
Otherwise, returns x, rounded to single-precision using the rounding mode specified in RN, in single-precision floating-point format.

ox_flag is set to 1 if rounding x resulted in an Overflow exception.
uxl_flag is set to 1 if rounding x resulted in an Underflow exception.
xxl_flag is set to 1 if rounding x returns an inexact result.
inc_flag is set to 1 if the significand of the result was incremented during rounding.
ConvertDPtoSP_NS(x)

is a single-precision floating-point value represented in double-precision format.

Returns x in single-precision format.

\[
\begin{align*}
\text{sign} & \leftarrow x.\text{bit}[0] \\
\text{exponent} & \leftarrow x.\text{bit}[1:11] \\
\text{fraction} & \leftarrow 0b1 \gg x.\text{bit}[12:63] \quad \text{// implicit bit set to 1 (for now)}
\end{align*}
\]

if (exponent == 0) & (fraction.\text{bit}[1:52] != 0) then do // DP Denormal operand
    exponent \leftarrow 0b000_0000_0001 // exponent override to DP Emin - 1
    fraction.\text{bit}[0] \leftarrow 0b0 // implicit bit override to 0
end

if (exponent < 897) && (fraction != 0) then do // SP tiny operand
    fraction \leftarrow fraction >>ui (897 - exponent) // denormalize until exponent = SP Emin
    exponent \leftarrow 0b011_1000_0000 // exponent override to SP Emin-1 = 896
end

return(sign » exponent.\text{bit}[0] » exponent.\text{bit}[4:10] » fraction.\text{bit}[1:23])

ConvertDPtoSW(x)

is a floating-point value in double-precision format.

If x is a NaN,
- \text{vxcvi\_flag} is set to 1,
- \text{vxsnan\_flag} is set to 1 if x is an SNaN, and
- return 0x8000_0000,

Otherwise, do the following.
Let \text{rnd} be the value x truncated to an integral value.

If \text{rnd} is greater than \(2^{23}-1\),
- \text{vxcvi\_flag} is set to 1,
- return 0x7FFF_FFFF.

Otherwise, if \text{rnd} is less than \(-2^{23}\),
- \text{vxcvi\_flag} is set to 1,
- return 0x8000_0000.

Otherwise,
- \text{xx\_flag} is set to 1 if \text{rnd} is inexact.
- return \text{rnd} in 32-bit signed integer format.

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.
ConvertDPtoUD(x)

x is a floating-point value in double-precision format.

If x is a NaN,
\( \text{vxcvi_flag} \) is set to 1,
\( \text{vxsnan_flag} \) is set to 1 if x is an SNaN, and
return 0x8000_0000_0000_0000,

Otherwise, do the following.
Let \( \text{rnd} \) be the value x truncated to an integral value.

If \( \text{rnd} \) is greater than \( 2^{64} - 1 \),
\( \text{vxcvi_flag} \) is set to 1,
return 0xFFFF_FFFF_FFFF_FFFF.

Otherwise, if \( \text{rnd} \) is less than 0,
\( \text{vxcvi_flag} \) is set to 1,
return 0x0000_0000_0000_0000.

Otherwise,
\( \text{xx_flag} \) is set to 1 if \( \text{rnd} \) is inexact.
return \( \text{rnd} \) in 64-bit unsigned integer format.

ConvertDPtoUW(x)

x is a floating-point value in double-precision format.

If x is a NaN,
\( \text{vxcvi_flag} \) is set to 1,
\( \text{vxsnan_flag} \) is set to 1 if x is an SNaN, and
return 0x0000_0000,

Otherwise, do the following.
Let \( \text{rnd} \) be the value x truncated to an integral value.

If \( \text{rnd} \) is greater than \( 2^{32} - 1 \),
\( \text{vxcvi_flag} \) is set to 1,
return 0xFFFF_FFFF.

Otherwise, if \( \text{rnd} \) is less than 0,
\( \text{vxcvi_flag} \) is set to 1,
return 0x0000_0000.

Otherwise,
\( \text{xx_flag} \) is set to 1 if \( \text{rnd} \) is inexact.
return \( \text{rnd} \) in 32-bit unsigned integer format.

ConvertFPtoDP(x)

Return the floating-point value x in DP format.

ConvertFPtoSP(x)

Return the floating-point value x in single-precision format.

ConvertSDtoFP(x)

x is a 64-bit signed integer value.
Return the value x converted to floating-point format having unbounded significand precision.
ConvertSPtoDP_NS(x)

x is a single-precision floating-point value.

Returns x in double-precision format.

```
sign ← x.bit[0]
fraction ← 0b0 » x.bit[9:31] » 0b0_0000_0000_0000_0000_0000_0000_0000
if (x.bit[1:8] == 255) then do // Infinity or NaN operand
    exponent ← 2047 // override exponent to DP Emax+1
end
else if (x.bit[1:8] == 0) && (fraction == 0) then do // SP Zero operand
    exponent ← 0 // override exponent to DP Emin-1
end
else if (x.bit[1:8] == 0) && (fraction != 0) then do // SP Denormal operand
    exponent ← 897 // override exponent to SP Emin
    do while (fraction.bit[0] == 0)
        fraction ← fraction << 1
        exponent ← exponent + 1
    end
end
return(sign » exponent » fraction.bit[1:52])
```
ConvertSP64toSP(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 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.

\[
\begin{align*}
\text{sign} & \leftarrow x.\text{bit}[0] \\
\text{exp} & \leftarrow x.\text{bit}[1:11] - 1023 \\
\text{frac} & \leftarrow x.\text{bit}[12:63] \\
\end{align*}
\]

if (exp = -1023) & (frac = 0) & (sign=0) then return(0x0000_0000) // +Zero
else if (exp = -1023) & (frac = 0) & (sign=1) then return(0x8000_0000) // -Zero
else if (exp = -1023) & (frac != 0) then return(0xUUUU_UUUU) // DP denorm
else if (exp < -126) then do // denormalization required
    \[
    \begin{align*}
    \text{msb} & \leftarrow 1 \\
    \text{do while (exp < -126)} & // denormalize operand until exp=Emin
    \text{frac.\text{bit}[1:51]} & \leftarrow \text{frac.\text{bit}[0:50]} \\
    \text{frac.\text{bit}[0]} & \leftarrow \text{msb} \\
    \text{msb} & \leftarrow 0 \\
    \text{exp} & \leftarrow \text{exp} + 1 \\
    \end{align*}
    \]
    if (frac = 0) then return(0xUUUU_UUUU) // value not representable in SP format
else do // return denormal SP
    \[
    \begin{align*}
    \text{result.\text{bit}[0]} & \leftarrow \text{sign} \\
    \text{result.\text{bit}[1:8]} & \leftarrow 0 \\
    \text{result.\text{bit}[9:31]} & \leftarrow \text{frac.\text{bit}[0:22]} \\
    \end{align*}
    \]
    return(result)
end
else if (exp = +1024) & (frac = 0) & (sign=0) then return(0x7F80_0000) // +Infinity
else if (exp = +1024) & (frac = 0) & (sign=1) then return(0xFF80_0000) // -Infinity
else if (exp = +1024) & (frac != 0) then do // QNaN or SNaN
    \[
    \begin{align*}
    \text{result.\text{bit}[0]} & \leftarrow \text{sign} \\
    \text{result.\text{bit}[1:8]} & \leftarrow 255 \\
    \text{result.\text{bit}[9:31]} & \leftarrow \text{frac.\text{bit}[0:22]} \\
    \end{align*}
    \]
    return(result)
end
else if (exp < +1024) & (exp > +126) then return(0xUUUU_UUUU) // overflow
else do // normal value
    \[
    \begin{align*}
    \text{result.\text{bit}[0]} & \leftarrow \text{sign} \\
    \text{result.\text{bit}[1:8]} & \leftarrow \text{exp.\text{bit}[4:11]} + 127 \\
    \text{result.\text{bit}[9:31]} & \leftarrow \text{frac.\text{bit}[0:22]} \\
    \end{align*}
    \]
    return(result)
end

ConvertSPtoDP(x)
x is a single-precision floating-point value.

If x is an SNaN, vxsnan_flag is set to 1.

If x is an SNaN, return x represented as a QNaN in double-precision floating-point format. Otherwise, if x is an QNaN, return x in double-precision floating-point format. Otherwise, return the value x in double-precision floating-point format.
ConvertSPtoSD(x)

x is a floating-point value in single-precision format.

If x is a NaN,
\vxcvi_flag\ is set to 1, and
\vxsnan_flag\ is set to 1 if x is an SNaN
return 0x8000_0000_0000_0000 and

Otherwise, do the following.
Let \( \text{round} \) be the value x truncated to an integral value.

If \( \text{round} \) is greater than \( 2^{63} - 1 \),
\vxcvi_flag\ is set to 1, and
return 0x7FFF_FFFF_FFFF_FFFF.

Otherwise, if \( \text{round} \) is less than \( -2^{63} \),
\vxcvi_flag\ is set to 1, and
return 0x8000_0000_0000_0000.

Otherwise,
\xx_flag\ is set to 1 if \( \text{round} \) is inexact, and
return \( \text{round} \) in 64-bit signed integer format.

ConvertSPtoSP64(x)

x is a floating-point value in single-precision format.

Returns the value x in double-precision format. If x is a SNaN, it is converted to a double-precision SNaN having the same payload as x.

\[ \text{sign} \leftarrow x.\text{bit}[0] \]
\[ \text{exp} \leftarrow x.\text{bit}[1:8] - 127 \]
\[ \text{frac} \leftarrow x.\text{bit}[9:31] \]

if (\( \text{exp} = -127 \)) \& \( (\text{frac} \neq 0) \) then do // Normalize the Denormal value
\[ \text{msb} \leftarrow \text{frac}.\text{bit}[0] \]
\[ \text{frac} \leftarrow \text{frac} \ll 1 \]
do while (\( \text{msb} = 0 \))
\[ \text{msb} \leftarrow \text{frac}.\text{bit}[0] \]
\[ \text{frac} \leftarrow \text{frac} \ll 1 \]
\[ \text{exp} \leftarrow \text{exp} - 1 \]
end
else if (\( \text{exp} = -127 \)) \& (\( \text{frac} = 0 \)) then exp \leftarrow -1023 // Zero value
else if (\( \text{exp} = +128 \)) then exp \leftarrow +1024 // Infinity, NaN

\[ \text{result}.\text{bit}[0] \leftarrow \text{sign} \]
\[ \text{result}.\text{bit}[1:11] \leftarrow \text{exp} + 1023 \]
\[ \text{result}.\text{bit}[12:34] \leftarrow \text{frac} \]
\[ \text{result}.\text{bit}[35:63] \leftarrow 0 \]
return(result)
ConvertSPtoSW(x)

x is a floating-point value in single-precision format.

If x is a NaN,
    vx_cvi_flag is set to 1,
    vx_snan_flag is set to 1 if x is an SNaN, and
    return 0x8000_0000.

Otherwise, do the following.
    Let rdn be the value x truncated to an integral value.

    If rdn is greater than $2^{31} - 1$,
        vx_cvi_flag is set to 1, and
        return 0x7FFFF_FFFF.

    Otherwise, if rdn is less than $-2^{31}$,
        vx_cvi_flag is set to 1, and
        return 0x8000_0000.

    Otherwise,
        xx_flag is set to 1 if rdn is inexact, and
        return rdn in 32-bit signed integer format.

ConvertSPtoUD(x)

x is a floating-point value in single-precision format.

If x is a NaN,
    vx_cvi_flag is set to 1, and
    vx_snan_flag is set to 1 if x is an SNaN
    return 0x0000_0000_0000_0000.

Otherwise, do the following.
    Let rdn be the value x truncated to an integral value.

    If rdn is greater than $2^{64} - 1$,
        vx_cvi_flag is set to 1, and
        return 0xFFFF_FFFF_FFFF_FFFF.

    Otherwise, if rdn is less than 0,
        vx_cvi_flag is set to 1, and
        return 0x0000_0000_0000_0000.

    Otherwise,
        xx_flag is set to 1 if rdn is inexact, and
        return rdn in 64-bit unsigned integer format.
ConvertSPtoUW(x)
  x is a floating-point value in single-precision format.

  If x is a NaN,
    vxvci_flag is set to 1,
    vxsnan_flag is set to 1 if x is an SNaN, and
    return 0x0000_0000.
  Otherwise, do the following.
    Let \( \text{rnd} \) be the value \( x \) truncated to an integral value.
    If \( \text{rnd} \) is greater than \( 2^{32}-1 \),
      vxvci_flag is set to 1, and
      return 0xFFFF_FFFF.
    Otherwise, if \( \text{rnd} \) is less than 0,
      vxvci_flag is set to 1, and
      return 0x0000_0000.
    Otherwise,
      xx_flag is set to 1 if \( \text{rnd} \) is inexact, and
      return \( \text{rnd} \) in 32-bit unsigned integer format.

ConvertSWtoFP(x)
  x is a 32-bit signed integer value.
  Return the value x converted to floating-point format having unbounded significand precision.

ConvertUDtoFP(x)
  x is a 64-bit unsigned integer value.
  Return the value x converted to floating-point format having unbounded significand precision.

ConvertUWtoFP(x)
  x is a 32-bit unsigned integer value.
  Return the value x converted to floating-point format having unbounded significand precision.

DivideDP(x, y)
  x and y are double-precision floating-point values.

  If x or y is an SNaN, vxsnan_flag is set to 1.
  If x is a Zero and y is a Zero, vxzdz_flag is set to 1.
  If x is a finite, nonzero value and y is a Zero, zx_flag is set to 1.
  If x is an Infinity and y is an Infinity, vxidi_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 a Zero and y is a Zero, return the standard QNaN.
  Otherwise, if x is a finite, nonzero value and y is a Zero with the same sign as x, return +Infinity.
  Otherwise, if x is a finite, nonzero value and y is a Zero with the opposite sign as x, return -Infinity.
  Otherwise, if x is an Infinity and y is an Infinity, return the standard QNaN.
  Otherwise, return the normalized quotient of x divided by y, having unbounded range and precision.
DivideSP(x, y)
  x and y are single-precision floating-point values.
  
  If x or y is an SNaN, vxsnan_flag is set to 1.
  
  If x is a Zero and y is a Zero, vxzd_flag is set to 1.
  
  If x is a finite, nonzero value and y is a Zero, zx_flag is set to 1.
  
  If x is an Infinity and y is an Infinity, vxidi_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 a Zero and y is a Zero, return the standard QNaN.
  Otherwise, if x is a finite, nonzero value and y is a Zero with the same sign as x, return +Infinity.
  Otherwise, if x is a finite, nonzero value and y is a Zero with the opposite sign as x, return -Infinity.
  Otherwise, if x is an Infinity and y is an Infinity, return the standard QNaN.
  Otherwise, return the normalized quotient of x divided by y, having unbounded range and precision.

DenormDP(x)
  x is a floating-point value having unbounded range and precision.
  
  Return the value x with its significand shifted right by a number of bits equal to the difference of the -1022 and the unbiased exponent of x, and its unbiased exponent set to -1022.

DenormSP(x)
  x is a floating-point value having unbounded range and precision.
  
  Return the value x with its significand shifted right by a number of bits equal to the difference of the -126 and the unbiased exponent of x, and its unbiased exponent set to -126.

EXTZ32(x)
  Result of extending the b-bit value x on the left with 32-b zeros, forming a 32-bit value.
  
  b ← LENGTH(x)
  result.bit[0:31-b] ← 0
  result.bit[32-b:31] ← x

EXTZ64(x)
  Result of extending the b-bit value x on the left with 64-b zeros, forming a 64-bit value.
  
  b ← LENGTH(x)
  result.bit[0:63-b] ← 0
  result.bit[64-b:63] ← x

EXTZ128(x)
  Result of extending the b-bit value x on the left with 128-b zeros, forming a 128-bit value.
  
  b ← LENGTH(x)
  result.bit[0:127-b] ← 0
  result.bit[128-b:127] ← x
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 0b10010 if x is a negative zero.
  Return 0b00010 if x is a positive zero.
  Return 0b11000 if x is a negative denormal value when represented in half-precision format.
  Return 0b10100 if x is a positive denormal value when represented in half-precision format.
  Return 0b01000 if x is a negative normal value when represented in half-precision format.
  Return 0b00100 if x is a positive normal value when represented in half-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 0b10010 if x is a negative zero.
  Return 0b00010 if x is a positive zero.
  Return 0b11000 if x is a negative denormal value when represented in double-precision format.
  Return 0b10100 if x is a positive denormal value when represented in double-precision format.
  Return 0b01000 if x is a negative normal value when represented in double-precision format.
  Return 0b00100 if x is a positive normal value when 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 0b01000 if x is negative and a normal number.
  Return 0b11000 if x is negative and a denormal number.
  Return 0b10010 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.
  Return 0b00101 if x is positive and an infinity.

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.
MaximumDP(x,y)
  x and y are double-precision floating-point values.

  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.

MaximumSP(x,y)
  x and y are single-precision floating-point values.

  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.

MinimumDP(x,y)
  x and y are double-precision floating-point values.

  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.

MinimumSP(x,y)
  x and y are single-precision floating-point values.

  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.
MultiplyAddDP(x,y,z)
  x, y and z are double-precision floating-point values.

  If x, y or z is an SNaN, vxsnan_flag is set to 1.

  If x is a Zero and y, is an Infinity or x is an Infinity and y is an Zero, vximz_flag is set to 1.

  If the product of x and y is an Infinity and z 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 z is an SNaN, return z.
  Otherwise, if y is a QNaN, return y.
  Otherwise, if y is an SNaN, return y represented as a QNaN.
  Otherwise, if x is a Zero and y is an Infinity or x is an Infinity and y is an Zero, return the standard QNaN.
  Otherwise, if the product of x and y is an Infinity, and z is an Infinity of the opposite sign, return the standard QNaN.
  Otherwise, return the normalized sum of z and the product of x and y, having unbounded range and precision.

MultiplyAddSP(x,y,z)
  x, y and z are single-precision floating-point values.

  If x, y or z is an SNaN, vxsnan_flag is set to 1.

  If x is a Zero and y is an Infinity, or x is an Infinity and y is an Zero, vximz_flag is set to 1.

  If the product of x and y is an Infinity and z 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 z is an SNaN, return z.
  Otherwise, if y is a QNaN, return y.
  Otherwise, if y is an SNaN, return y represented as a QNaN.
  Otherwise, if x is a Zero and y is an Infinity or x is an Infinity and y is an Zero, return the standard QNaN.
  Otherwise, if the product of x and y is an Infinity, and z is an Infinity of the opposite sign, return the standard QNaN.
  Otherwise, return the normalized sum of z and the product of x and y, having unbounded range and precision.

MultiplyDP(x,y)
  x and y are double-precision floating-point values.

  If x or y is an SNaN, vxsnan_flag is set to 1.

  If x is a Zero and y is an Infinity, or x is an Infinity and y is an Zero, vximz_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 a Zero and y is an Infinity or x is an Infinity and y is an Zero, return the standard QNaN.
  Otherwise, return the normalized product of x and y, having unbounded range and precision.
MultiplySP(x,y)
  x and y are single-precision floating-point values.

  If x or y is an SNaN, vxsnan_flag is set to 1.

  If x is a Zero and y is an Infinity, or x is an Infinity and y is an Zero, vximz_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 a Zero and y is an Infinity or x is an Infinity and y is an Zero, return the standard QNaN.
  Otherwise, return the normalized product of x and y, having unbounded range and precision.

NegateDP(x)
  If the double-precision floating-point value x is a NaN, return x.
  Otherwise, return the double-precision floating-point value x with its sign bit complemented.

NegateSP(x)
  If the single-precision floating-point value x is a NaN, return x.
  Otherwise, return the single-precision floating-point value x with its sign bit complemented.

ReciprocalEstimateDP(x)
  x is a double-precision floating-point value.

  If x is an SNaN, vxsnan_flag is set to 1.

  If x is a Zero, 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 x is a Zero, return an Infinity with the sign of x.
  Otherwise, if x is an Infinity, return a Zero with the sign of x.
  Otherwise, return an estimate of the reciprocal of x having unbounded exponent range.

ReciprocalEstimateSP(x)
  x is a single-precision floating-point value.

  If x is an SNaN, vxsnan_flag is set to 1.

  If x is a Zero, 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 x is a Zero, return an Infinity with the sign of x.
  Otherwise, if x is an Infinity, return a Zero with the sign of x.
  Otherwise, return an estimate of the reciprocal of x having unbounded exponent range.
ReciprocalSquareRootEstimateDP(x)
  x is a double-precision floating-point value.
  
  If x is an SNaN, `vxsnan_flag` is set to 1.
  
  If x is a Zero, `zx_flag` is set to 1.
  
  If x is a negative, nonzero number, `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 a negative, nonzero value, return the default QNaN.
  Otherwise, return an estimate of the reciprocal of the square root of x having unbounded exponent range.

ReciprocalSquareRootEstimateSP(x)
  x is a single-precision floating-point value.
  
  If x is an SNaN, `vxsnan_flag` is set to 1.
  
  If x is a Zero, `zx_flag` is set to 1.
  
  If x is a negative, nonzero number, `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 a negative, nonzero value, return the default QNaN.
  Otherwise, return an estimate of the reciprocal of the square root of x having unbounded exponent range.

reset_xflags()
  `vxsnan_flag` is set to 0.
  `vximz_flag` is set to 0.
  `vxidi_flag` is set to 0.
  `vxisi_flag` is set to 0.
  `vzxz_flag` is set to 0.
  `vxsqrt_flag` is set to 0.
  `vxcvi_flag` is set to 0.
  `vxvc_flag` is set to 0.
  `ox_flag` is set to 0.
  `ux_flag` is set to 0.
  `xx_flag` is set to 0.
  `zx_flag` is set to 0.
RoundToDP(x, y)

x is a 2-bit unsigned integer specifying one of four rounding modes.

- 0b00: Round to Nearest Even
- 0b01: Round towards Zero
- 0b10: Round towards +Infinity
- 0b11: Round towards -Infinity

y is a normalized floating-point value having unbounded range and precision.

Return the value y rounded to double-precision under control of the rounding mode specified by x.

```plaintext
if isQNan(y) then return ConvertFPtoDP(y)
if isInf(y) then return ConvertFPtoDP(y)
if isZero(y) then return ConvertFPtoDP(y)
if y < Nmin then do
  if UE = 0 then do
    if x = 0b00 then r ← RoundToDPNearEven(DenormDP(y))
    if x = 0b01 then r ← RoundToDPTrunc(DenormDP(y))
    if x = 0b10 then r ← RoundToDPCeil(DenormDP(y))
    if x = 0b11 then r ← RoundToDPFloor(DenormDP(y))
    ux_flag ← xx_flag
    return(ConvertFPtoDP(r))
  end
  else do
    y ← Scalb(y, +1536)
    ux_flag ← 1
  end
end
if x = 0b00 then r ← RoundToDPNearEven(y)
if x = 0b01 then r ← RoundToDPTrcunc(y)
if x = 0b10 then r ← RoundToDPCeil(y)
if x = 0b11 then r ← RoundToDPFloor(y))
if r > Nmax then do
  if OE = 0 then do
    if x = 0b00 then r ← sign ? -Inf : +Inf
    if x = 0b01 then r ← sign ? -Nmax : +Nmax
    if x = 0b10 then r ← sign ? -Nmax : +Inf
    if x = 0b11 then r ← sign ? -Inf : +Nmax
    ox_flag ← 0b1
    inc_flag ← 0bU
    return(ConvertFPtoDP(r))
  end
  else do
    r ← Scalb(r, -1536)
    ox_flag ← 1
  end
end
return(ConvertFPtoDP(r))
```
RoundToDPCeil(x)

x is a floating-point value having unbounded range and precision.

If x is a QNaN, return x.

Otherwise, if x is an Infinity, return x.

Otherwise, do the following.

Return the smallest floating-point number having unbounded exponent range but double-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.

RoundToDPFloor(x)

x is a floating-point value having unbounded range and precision.

If x is a QNaN, return x.

Otherwise, if x is an Infinity, return x.

Otherwise, do the following.

Return the largest floating-point number having unbounded exponent range but double-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.

RoundToDPIntegerCeil(x)

x is a double-precision floating-point value.

If x is an SNaN, vxssnan_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 an infinity, return x.

Otherwise, do the following.

Return the smallest double-precision floating-point integer value 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.
RoundToDPIntegerFloor(x)
  x is a double-precision floating-point value.

  If x is an SNaN, vxsnan_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 an infinity, return x.
  Otherwise, do the following.
    Return the largest double-precision floating-point integer value 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.

RoundToDPIntegerNearAway(x)
  x is a double-precision floating-point value.

  If x is an SNaN, vxsnan_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 an infinity, return x.
  Otherwise, do the following.
    Return the largest double-precision floating-point integer value that is lesser or equal in value to x+0.5 if x>0, or the smallest double-precision floating-point integer that is greater or equal in value to x-0.5 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.

RoundToDPIntegerNearEven(x)
  x is a double-precision floating-point value.

  If x is an SNaN, vxsnan_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 an infinity, return x.
  Otherwise, do the following.
    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 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.
RoundToDPIntegerTrunc(x)
   x is a double-precision floating-point value.

   If x is an SNaN, vxsnan_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 an infinity, return x.

   Otherwise, do the following.
      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 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.

RoundToDPNearEven(x)
   x is a floating-point value having unbounded range and precision.

   If x is a QNaN, return x.

   Otherwise, if x is an Infinity, return x.

   Otherwise, do the following.
      Return the floating-point number having unbounded exponent range but double-precision significand precision that is nearest in value to x (in case of a tie, the floating-point number having unbounded exponent range but double-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.

RoundToDPTrunc(x)
   x is a floating-point value having unbounded range and precision.

   If x is a QNaN, return x.

   Otherwise, if x is an Infinity, return x.

   Otherwise, do the following.
      Return the largest floating-point number having unbounded exponent range but double-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 double-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.
RoundToSP(x,y)
 x is a 2-bit unsigned integer specifying one of four rounding modes.

<table>
<thead>
<tr>
<th>x</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 towards Zero</td>
</tr>
<tr>
<td>0b10</td>
<td>Round towards +Infinity</td>
</tr>
<tr>
<td>0b11</td>
<td>Round towards -Infinity</td>
</tr>
</tbody>
</table>

y is a normalized floating-point value having unbounded range and precision.

Return the value y rounded to single-precision under control of the rounding mode specified by x.

if IsQNaN(y) then return ConvertFPtoSP(y)
if IsInf(y) then return ConvertFPtoSP(y)
if IsZero(y) then return ConvertFPtoSP(y)
if y<Nmin then do
  if UE=0 then do
    if x=0b00 then r \leftarrow RoundToSPNearEven( DenormSP(y) )
    if x=0b01 then r \leftarrow RoundToSPTrunc( DenormSP(y) )
    if x=0b10 then r \leftarrow RoundToSPCeil( DenormSP(y) )
    if x=0b11 then r \leftarrow RoundToSPFloor( DenormSP(y) )
    ux_flag \leftarrow xx_flag
    return(ConvertFPtoSP(r))
  end
  else do
    y \leftarrow Scalb(y, +192)
    ux_flag \leftarrow 1
  end
end
if x=0b00 then r \leftarrow RoundToSPNearEven(y)
if x=0b01 then r \leftarrow RoundToSPTrunc(y)
if x=0b10 then r \leftarrow RoundToSPCeil(y)
if x=0b11 then r \leftarrow RoundToSPFloor(y))
if r>Nmax then do
  if OE=0 then do
    if x=0b00 then r \leftarrow sign ? -Inf : +Inf
    if x=0b01 then r \leftarrow sign ? -Nmax : +Nmax
    if x=0b10 then r \leftarrow sign ? -Nmax : +Inf
    if x=0b11 then r \leftarrow sign ? -Inf : +Nmax
    ox_flag \leftarrow 0b1
    inc_flag \leftarrow 0bU
    return(ConvertFPtoSP(r))
  end
  else do
    r \leftarrow Scalb(r, -192)
    ox_flag \leftarrow 1
  end
end
return(ConvertFPtoSP(r))
RoundToSPCeil(x)
  x is a floating-point value having unbounded range and precision.
  
  If x is a QNaN, return x.
  Otherwise, if x is an Infinity, return x.
  
  Otherwise, do the following.
  Return the smallest floating-point number having unbounded exponent range but single-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.

RoundToSPFloor(x)
  x is a floating-point value having unbounded range and precision.
  
  If x is a QNaN, return x.
  Otherwise, if x is an Infinity, return x.
  
  Otherwise, do the following.
  Return the largest floating-point number having unbounded exponent range but single-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.

RoundToSPIntegerCeil(x)
  x is a single-precision floating-point value.
  
  If x is an SNaN, vxsnan_flag is set to 1.
  If x is a QNaN, return x.
  
  Otherwise, if x is a SNaN, return x represented as a QNaN.
  
  Otherwise, if x is an infinity, return x.
  
  Otherwise, do the following.
  Return the smallest single-precision floating-point integer value 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.
RoundToSPIntegerFloor(x)
  x is a single-precision floating-point value.

  If x is an SNaN, \texttt{vxsnan\_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 an infinity, return x.
  Otherwise, do the following.
    Return the largest single-precision floating-point integer value that is lesser or equal in value to x.
    If the magnitude of the value returned is greater than x, \texttt{inc\_flag} is set to 1.
    If the value returned is not equal to x, \texttt{xx\_flag} is set to 1.

RoundToSPIntegerNearAway(x)
  x is a single-precision floating-point value.

  If x is an SNaN, \texttt{vxsnan\_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 an infinity, return x.
  Otherwise, do the following.
    Return x if x is a floating-point integer; otherwise return the largest single-precision floating-point integer value that is lesser or equal in value to x+0.5 if x>0, or the smallest single-precision floating-point integer value that is greater or equal in value to x-0.5 if x<0.
    If the magnitude of the value returned is greater than x, \texttt{inc\_flag} is set to 1.
    If the value returned is not equal to x, \texttt{xx\_flag} is set to 1.

RoundToSPIntegerNearEven(x)
  x is a single-precision floating-point value.

  If x is an SNaN, \texttt{vxsnan\_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 an infinity, return x.
  Otherwise, do the following.
    Return x if x is a floating-point integer; otherwise return the single-precision floating-point integer value that is nearest in value to x (in case of a tie, the single-precision floating-point integer value with the least-significant bit equal to 0 is used).
    If the magnitude of the value returned is greater than x, \texttt{inc\_flag} is set to 1.
    If the value returned is not equal to x, \texttt{xx\_flag} is set to 1.
**RoundToSPIntegerTrunc(x)**

x is a single-precision floating-point value.

If x is a QNaN, return x.

Otherwise, if x is an SNaN, return x represented as a QNaN, and vxsnan_flag is set to 1.

Otherwise, if x is an infinity, return x.

Otherwise, do the following.

Return the largest single-precision floating-point integer value that is lesser or equal in value to x if x>0, or the smallest single-precision floating-point integer value 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.

**RoundToSPNearEven(x)**

x is a floating-point value having unbounded range and precision.

If x is a QNaN, return x.

Otherwise, if x is an Infinity, return x.

Otherwise, do the following.

Return the floating-point number having unbounded exponent range but single-precision significand precision that is nearest in value to x (in case of a tie, the floating-point number having unbounded exponent range but single-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.

**RoundToSPTrunc(x)**

x is a floating-point value having unbounded range and precision.

If x is a QNaN, return x.

Otherwise, if x is an Infinity, return x.

Otherwise, do the following.

Return the largest floating-point number having unbounded exponent range but single-precision significand precision that is lesser or equal in value to x if x>0, or the smallest single-precision floating-point number 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.

**Scalb(x,y)**

x is a floating-point value having unbounded range and precision.

y is a signed integer.

Result of multiplying the floating-point value x by $2^y$.

**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.
SquareRootDP(x)
   x is a double-precision floating-point value.
   If x is an SNaN, vxsnan_flag is set to 1.
   If x is a negative, nonzero value, 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 a negative, nonzero value, return the default QNaN.
   Otherwise, return the normalized square root of x, having unbounded range and precision.

SquareRootSP(x)
   x is a single-precision floating-point value.
   If x is an SNaN, vxsnan_flag is set to 1.
   If x is a negative, nonzero value, 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 a negative, nonzero value, return the default QNaN.
   Otherwise, return the normalized square root of x, having unbounded range and precision.
7.6.3 VSX Instruction Descriptions

**Load VSX Scalar Doubleword DS-form**

\[
\text{lxsd VRT,DS(RA)}
\]

\[
\begin{array}{cccccc}
0 & 57 & 6 & 11 & 16 & 2 \\
\end{array}
\]

if MSR.VEC=0 then Vector_Unavailable()

\[
\begin{align*}
\text{EA} & \leftarrow ((\text{RA}=0) \ ? \ 0 : \ GPR[\text{RA}]) + \text{EXTS(} \text{DS} \text{)} \ll 2 \\
\text{VSR}[\text{VRT}+32].\text{dword}[0] & \leftarrow \text{MEM(EA,8)} \\
\text{VSR}[\text{VRT}+32].\text{dword}[1] & \leftarrow \text{0xUUUU_UUUU_UUUU_UUUU}
\end{align*}
\]

Let XT be the value VRT + 32.

Let EA be the sum of the contents of GPR[RA], or 0 if RA=0, and the signed integer value DS<<2.

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, let 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 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 0 of VSR[XT].

The contents of doubleword element 1 of VSR[XT] are undefined.

**Special Registers Altered:**

None

---

**Load VSX Scalar Doubleword Indexed X-form**

\[
\text{lxsdx XT,RA,RB}
\]

\[
\begin{array}{cccccccccc}
0 & 31 & 6 & 11 & 16 & 21 & 588 & 31 \\
\end{array}
\]

if MSR.VSX=0 then VSX_Unavailable()

\[
\begin{align*}
\text{EA} & \leftarrow ((\text{RA}=0) \ ? \ 0 : \ GPR[\text{RA}]) + GPR[\text{RB}] \\
\text{VSR}[32\times\text{TX}+T].\text{dword}[0] & \leftarrow \text{MEM(EA,8)} \\
\text{VSR}[32\times\text{TX}+T].\text{dword}[1] & \leftarrow \text{0xUUUU_UUUU_UUUU_UUUU}
\end{align*}
\]

Let XT be the value 32×TX + T.

Let EA be the sum of the contents of GPR[RA], or 0 if RA=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.

When Little-Endian byte ordering is employed, the contents of the doubleword in storage at address EA are placed into load_data such 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.

When Little-Endian byte ordering is employed, let 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 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 0 of VSR[XT].

The contents of doubleword element 1 of VSR[XT] are undefined.

**Special Registers Altered:**

None
### VSR Data Layout for lxsd

\[
tgt = VSR[XT]
\]

<table>
<thead>
<tr>
<th>MEM(EA,8)</th>
<th>undefined</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>64</td>
</tr>
</tbody>
</table>

### VSR Data Layout for lxsdx

\[
tgt = VSR[XT]
\]

<table>
<thead>
<tr>
<th>MEM(EA,8)</th>
<th>undefined</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>64</td>
</tr>
</tbody>
</table>
Load VSX Scalar as Integer Byte & Zero Indexed X-form

**lxsibzx XT,RA,RB**

<table>
<thead>
<tr>
<th>31</th>
<th>8</th>
<th>T</th>
<th>RA</th>
<th>RB</th>
<th>781</th>
<th>[X]</th>
</tr>
</thead>
</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]

VSR[32×TX+T].dword[0] ← EXTZ64(MEM(EA,1))
VSR[32×TX+T].dword[1] ← 0xUUUU_UUUU_UUUU_UUUU

Let XT be the value 32×TX + T.

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 unsigned integer in the byte in storage addressed by EA is placed in doubleword element 0 of VSR[XT]. The contents of doubleword element 1 of VSR[XT] are undefined.

**Special Registers Altered:**

None

---

**VSR Data Layout for lxsibzx**

tgt = VSR[XT]

<table>
<thead>
<tr>
<th>tgt.dword[0]</th>
<th>undefined</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>64</td>
</tr>
</tbody>
</table>

Load VSX Scalar as Integer Halfword & Zero Indexed X-form

**lxsihzx XT,RA,RB**

<table>
<thead>
<tr>
<th>31</th>
<th>8</th>
<th>T</th>
<th>RA</th>
<th>RB</th>
<th>813</th>
<th>[X]</th>
</tr>
</thead>
</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]

VSR[32×TX+T].dword[0] ← EXTZ64(MEM(EA,2))
VSR[32×TX+T].dword[1] ← 0xUUUU_UUUU_UUUU_UUUU

Let XT be the value 32×TX + T.

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 unsigned integer in the halfword in storage addressed by EA is placed in doubleword element 0 of VSR[XT]. The contents of doubleword element 1 of VSR[XT] are undefined.

**Special Registers Altered:**

None

---

**VSR Data Layout for lxsihzx**

tgt = VSR[XT]

<table>
<thead>
<tr>
<th>tgt.dword[0]</th>
<th>undefined</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>64</td>
</tr>
</tbody>
</table>
Load VSX Scalar as Integer Word Algebraic Indexed X-form

```plaintext
lxsiwax XT,RA,RB
```

- if MSR.VSX=0 then VSX_Unavailable()

EA ← ((RA=0) ? 0 : GPR[RA]) + GPR[RB]

VSR[32×TX+T].dword[0] ← EXT64(MEM(EA,4))
VSR[32×TX+T].dword[1] ← 0xUUUU_UUUU_UUUU_UUUU

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 0 of load_data,
- the contents of the byte in storage at address EA+1 are placed into byte 2 of load_data, and
- 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 sign-extended to a doubleword and placed in doubleword element 0 of VSR[XT].

The contents of doubleword element 1 of VSR[XT] are undefined.

Special Registers Altered

None
Load VSX Scalar as Integer Word and Zero Indexed X-form

lxsiwzx XT,RA,RB

0 31 12 21 16 11 6 T RA RB

if MSR.VSX=0 then VSX_Unavailable()

EA \leftarrow ((RA=0) ? 0 : GPR[RA]) + GPR[RB]

VSR[32\times TX + T].dword[0] \leftarrow \text{ExtendZero}(\text{MEM}(EA, 4))

VSR[32\times TX + T].dword[1] \leftarrow 0xUUUU_UUUU_UUUU_UUUU

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 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 to a doubleword and placed in doubleword element 0 of VSR[XT].

The contents of doubleword element 1 of VSR[XT] are undefined.

Special Registers Altered

None
Load VSX Scalar Single DS-form

\textbf{lxssp} \hspace{1cm} \text{VRT, DS(RA)}

\begin{tabular}{|c|c|c|c|}
\hline
57 & VRT & RA & DS \\
\hline
0 & 6 & 11 & 16 \\
3 & 30 & 31 & 3 \\
\hline
\end{tabular}

\begin{itemize}
\item if MSR.VEC=0 then VectorUnavailable()
\end{itemize}

\begin{itemize}
\item EA \leftarrow (\text{RA=0} \iff 0 ) ? 0 : \text{GPR}[RA] + \text{EXTS}(DS||0b00)
\end{itemize}

\begin{itemize}
\item VSR[VRT+32].dword[0] \leftarrow \text{ConvertSPtoSP64(MEM(EA,4))}
\end{itemize}

\begin{itemize}
\item VSR[VRT+32].dword[1] \leftarrow 0xUUUU_UUUU_UUUU_UUUU
\end{itemize}

Let XT be the value VRT + 32.

Let EA be the sum of the contents of GPR[RA], or 0 if RA=0, and the signed integer value DS||0b00.

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;

\begin{itemize}
\item the contents of the byte in storage at address EA are placed into byte 0 of load_data,
\item the contents of the byte in storage at address EA+1 are placed into byte 1 of load_data,
\item the contents of the byte in storage at address EA+2 are placed into byte 2 of load_data, and
\item the contents of the byte in storage at address EA+3 are placed into byte 3 of load_data.
\end{itemize}

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;

\begin{itemize}
\item the contents of the byte in storage at address EA are placed into byte 3 of load_data,
\item the contents of the byte in storage at address EA+1 are placed into byte 2 of load_data,
\item the contents of the byte in storage at address EA+2 are placed into byte 1 of load_data, and
\item the contents of the byte in storage at address EA+3 are placed into byte 0 of load_data.
\end{itemize}

load_data, interpreted as a single-precision floating-point value, 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 undefined.

\textbf{Special Registers Altered:}

\begin{itemize}
\item None
\end{itemize}

Load VSX Scalar Single-Precision Indexed X-form

\textbf{lxsspx} \hspace{1cm} \text{XT, RA, RB}

\begin{tabular}{|c|c|c|c|c|c|}
\hline
31 & T & RA & RB & 524 & \% \\
\hline
0 & 6 & 11 & 16 & 21 & 31 \\
\hline
\end{tabular}

\begin{itemize}
\item if MSR.VSX=0 then VSXUnavailable()
\end{itemize}

\begin{itemize}
\item EA \leftarrow (\text{RA=0} \iff 0 ) ? 0 : \text{GPR}[RA] + \text{GPR}[RB]
\end{itemize}

\begin{itemize}
\item VSR[VRT+32].dword[0] \leftarrow \text{ConvertSPtoSP64(MEM(EA,4))}
\end{itemize}

\begin{itemize}
\item VSR[VRT+32].dword[1] \leftarrow 0xUUUU_UUUU_UUUU_UUUU
\end{itemize}

Let XT be the value 32\times T + 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;

\begin{itemize}
\item the contents of the byte in storage at address EA are placed into byte 0 of load_data,
\item the contents of the byte in storage at address EA+1 are placed into byte 1 of load_data,
\item the contents of the byte in storage at address EA+2 are placed into byte 2 of load_data, and
\item the contents of the byte in storage at address EA+3 are placed into byte 3 of load_data.
\end{itemize}

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;

\begin{itemize}
\item the contents of the byte in storage at address EA are placed into byte 3 of load_data,
\item the contents of the byte in storage at address EA+1 are placed into byte 2 of load_data,
\item the contents of the byte in storage at address EA+2 are placed into byte 1 of load_data, and
\item the contents of the byte in storage at address EA+3 are placed into byte 0 of load_data.
\end{itemize}

load_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 undefined.

\textbf{Special Registers Altered}

\begin{itemize}
\item None
\end{itemize}
### VSR Data Layout for lxssp

\[ \text{tgt} = \text{VSR}[\text{XT}] \]

<table>
<thead>
<tr>
<th></th>
<th>DP</th>
<th>undefined</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>64</td>
<td></td>
</tr>
<tr>
<td>64</td>
<td></td>
<td>127</td>
</tr>
</tbody>
</table>

### VSR Data Layout for lxssp

\[ \text{tgt} = \text{VSR}[\text{XT}] \]

<table>
<thead>
<tr>
<th></th>
<th>DP</th>
<th>undefined</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>64</td>
<td></td>
</tr>
<tr>
<td>64</td>
<td></td>
<td>127</td>
</tr>
</tbody>
</table>
**Load VSX Vector Byte*16 Indexed X-form**

```
lxvb16x XT,RA,RB
```

<table>
<thead>
<tr>
<th>31</th>
<th>8</th>
<th>11</th>
<th>10</th>
<th>1</th>
<th>876</th>
<th>7</th>
</tr>
</thead>
</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]

do i = 0 to 15
   VSR[32×TX+T].byte[i] ← MEM(EA+i, 1)
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].

For each integer value from 0 to 15, do the following.
   The contents of the byte in storage at address EA+i are placed into byte element i of VSR[XT].

Special Registers Altered:
   None

---

### Programming Note

- `lxvd2x`, `lxvw4x`, `lxvh8x`, `lxvb16x`, and `lxvx` exhibit identical behavior in Big-Endian mode.

---

Example: Loading data using Load VSX Vector Byte*16 Indexed

```c
char X[16] = { 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
        0 1 2 3 4 5 6 7 8 9 A B C D E F
```

**Little-endian storage image of X**

```
addr(X): F0 F1 F2 F3 F4 F5 F6 F7 E0 E1 E2 E3 E4 E5 E6 E7
        0 1 2 3 4 5 6 7 8 9 A B C D E F
```

Loading a vector of 16 byte elements from Big-Endian storage in VSR[XT] using `lxvb16x`, retaining left-to-right element ordering.

```c
# Assumptions
# GPR[PX] = address of X
lxvb16x xX, r0, rPX
```

**VSR(W):**

```
F0 F1 F2 F3 F4 F5 F6 F7 E0 E1 E2 E3 E4 E5 E6 E7
        0 1 2 3 4 5 6 7 8 9 A B C D E F
```

Loading a vector of 16 byte elements from Little-Endian storage in VSR[XT] using `lxvb16x`, retaining left-to-right element ordering.

```c
# Assumptions
# GPR[PX] = address of X
lxvb16x xX, r0, rPX
```

**VSR(X):**

```
F0 F1 F2 F3 F4 F5 F6 F7 E0 E1 E2 E3 E4 E5 E6 E7
        0 1 2 3 4 5 6 7 8 9 A B C D E F
```

---

Chapter 7. Vector-Scalar Floating-Point Operations 487
Load VSX Vector Doubleword*2 Indexed X-form

\texttt{lxvd2x} XT,RA,RB

\begin{center}
\begin{tabular}{|c|c|c|c|c|}
\hline
\textbf{T} & \textbf{RA} & \textbf{RB} & \textbf{844} & \textbf{X} \\
\hline
31 & 6 & 21 & 21 & 31 \\
\hline
\end{tabular}
\end{center}

\begin{itemize}
\item If MSR.VSX=0 then VSX.Unavailable()
\item \( EA \leftarrow ((RA=0) \ ? 0 : \text{GPR}[RA]) + \text{GPR}[RB] \)
\item \( \text{VSR}[32\times T+T].\text{dword}[0] \leftarrow \text{MEM}(EA, 8) \)
\item \( \text{VSR}[32\times T+T].\text{dword}[1] \leftarrow \text{MEM}(EA+8, 8) \)
\end{itemize}

Let \( XT \) be the value \( 32\times T + T \).

Let \( EA \) be the sum of the contents of \( \text{GPR}[RA] \), or 0 if \( RA \) is equal to 0, and the contents of \( \text{GPR}[RB] \).

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 \( EA+8\times i \) are placed into \texttt{load\_data} in such an order that:

- the contents of the byte in storage at address \( EA+8\times i \) are placed into byte element 0 of \texttt{load\_data},
- the contents of the byte in storage at address \( EA+8\times i+1 \) are placed into byte element 1 of \texttt{load\_data}, and so forth until
- the contents of the byte in storage at address \( EA+8\times i+7 \) are placed into byte element 7 of \texttt{load\_data}.

When Little-Endian byte ordering is employed, the contents of the doubleword in storage at address \( EA+8\times i \) are placed into \texttt{load\_data} in such an order that:

- the contents of the byte in storage at address \( EA+8\times i \) are placed into byte element 7 of \texttt{load\_data},
- the contents of the byte in storage at address \( EA+8\times i+1 \) are placed into byte element 6 of \texttt{load\_data}, and so forth until
- the contents of the byte in storage at address \( EA+8\times i+7 \) are placed into byte element 0 of \texttt{load\_data}.

\texttt{load\_data} is placed into doubleword element \( i \) of \texttt{VSR}[XT].

Special Registers Altered

None

VSR Data Layout for \texttt{lxvd2x}

\begin{tabular}{|c|c|}
\hline
\textbf{.dword[0]} & \textbf{.dword[1]} \\
\hline
0 & 64 & 127 \\
\hline
\end{tabular}

Programming Note

\texttt{lxvd2x}, \texttt{lxvw4x}, \texttt{lxvh8x}, \texttt{lxvb16x}, and \texttt{lxvx} exhibit identical behavior in Big-Endian mode.
Load VSX Vector with Length X-form

lxvl XT,RA,RB

<table>
<thead>
<tr>
<th>31</th>
<th>T</th>
<th>RA</th>
<th>RB</th>
<th>269</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()

EA \leftarrow (RA=0) ? 0 : GPR[RA]

nb \leftarrow EXTZ(GPR[RB].bit[0:7]) if nb>16 then nb \leftarrow 16

load_data \leftarrow 0x0000_0000_0000_0000_0000_0000_0000_0000

if MSR.LE = 0 then // Big-Endian byte-ordering

load_data.byte[0:nb-1] \leftarrow MEM(EA,nb)
else // Little-Endian byte-ordering

load_data.byte[16-nb:15] \leftarrow MEM(EA,nb)

VSR[32TX+T] \leftarrow 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
Example: Loading less than 16-byte data into VSR using lxvl

```
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>addr(S)+0x0010</th>
<th>addr(S)+0x0020</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
<td>E2 E3 E4 E5 E6 E7 E8 E9 EA EB F0 F1 F2 F3 F4 F5 F6</td>
<td>F6 F7 F8 F9</td>
</tr>
</tbody>
</table>

VSR register image of S, X, & Z

<table>
<thead>
<tr>
<th>VSR[S]</th>
<th>VSR[X]</th>
<th>VSR[Z]</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
<td>E0 E1 E2 E3 E4 E5 E6 E7 E8 EA EB</td>
<td>F0 F1 F2 F3 F4 F5 F6 F7 F8 F9</td>
</tr>
</tbody>
</table>

Loading less than 16-byte data from Little-Endian storage in VSR[XT] using `lxvl`.

Little-endian storage image of S, X, & Z

<table>
<thead>
<tr>
<th>addr(S)+0x0000</th>
<th>addr(S)+0x0010</th>
<th>addr(S)+0x0020</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
<td>E3 E2 E5 E4 E7 E6 E9 EB EA F8 F7 F6 F5 F4</td>
<td>F3 F2 F1 F0</td>
</tr>
</tbody>
</table>

VSR register image of S, X, & Z

<table>
<thead>
<tr>
<th>VSR[S]</th>
<th>VSR[X]</th>
<th>VSR[Z]</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
<td>E3 E2 E5 E4 E7 E6 E9 EB EA</td>
<td>F0 F1 F2 F3 F4 F5 F6 F7 F8 F9</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,rLS
lxvl xX,rPX,rLX
lxvl xZ,rPZ,rLZ
### Load VSX Vector Left-justified with Length X-form

**lxvll XT,RA,RB**

```markdown
<table>
<thead>
<tr>
<th>31</th>
<th>T</th>
<th>RA</th>
<th>RB</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
</table>
```

If \( TX = 0 \) & MSR.VSX=0 then VSX_Unavailable()
If \( TX = 1 \) & MSR.VEC=0 then Vector_Unavailable()

\[ EA \leftarrow \begin{cases} \text{RA=0 ? 0 : GPR[RA]} & \text{if} \ nb > 16 \text{ then } nb \leftarrow 16 \\ \text{EXTZ(GPR[RB].bit[0:7])} & \text{else} \end{cases} \]

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 \( nb \) is equal to 0, the storage access is not performed and the contents of VSR[XT] are set to 0.
- 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

---

### Example: Loading less than 16-byte left-justified data

**Decimal Values**

- \( X = +1234567890123456789 \)
- \( Y = -123456 \)
- \( Z = +1004966723510220 \)

**Initial state of VSRs X, Y, & Z**

| VSR[X] | FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF |
| VSR[Y] | FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF |
| VSR[Z] | FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF |

**Big-endian & Little-Endian storage image of X, Y, & Z**

| \( X+0x0000 \) | 12 34 56 78 90 12 34 56 78 9C 01 23 45 6D 01 00 |
| \( X+0x0010 \) | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |

**# 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

**lxvll xX,rPX,rNX**

**lxvll xY,rPY,rNY**

**lxvll xZ,rPZ,rNZ**

**Final state of VSRs X, Y, & Z**

| VSR[X] | 01 34 67 78 90 12 34 56 78 9C 00 00 00 00 00 00 |
| VSR[Y] | 01 23 45 6D 00 00 00 00 00 00 00 00 00 00 00 00 |
| VSR[Z] | 01 00 49 66 72 35 10 22 0C 00 00 00 00 00 00 00 |

---

**Chapter 7. Vector-Scalar Floating-Point Operations**
Load VSX Vector DQ-form

\[ \text{lxv} \ XT,\text{DQ}(\text{RA}) \]

- if \( TX=0 \) & \( \text{MSR.VSX}=0 \) then \( \text{VSX.Unavailable}() \)
- if \( TX=1 \) & \( \text{MSR.VEC}=0 \) then \( \text{Vector.Unavailable}() \)

\[ \text{EA} \leftarrow ((\text{RA}=0) ? 0 : \text{GPR}[\text{RA}]) + \text{EXTS}(\text{DQ}) || 0b0000 \]

\[ \text{VSR}[32\times TX+T] \leftarrow \text{MEM}(\text{EA},16) \]

Let \( XT \) be the value \( 32 \times TX + T \).

Let the effective address (EA) be the sum of the contents of \( \text{GPR}[\text{RA}] \), or 0 if \( \text{RA} \) is equal to 0, and the signed integer value \( \text{DQ} || 0b0000 \).

When Big-Endian byte ordering is employed, the contents of the quadword 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 element 0 of \( \text{load\_data} \),
- the contents of the byte in storage at address \( \text{EA}+1 \) are placed into byte element 1 of \( \text{load\_data} \), and so forth until
- the contents of the byte in storage at address \( \text{EA}+15 \) are placed into byte element 15 of \( \text{load\_data} \).

When Little-Endian byte ordering is employed, the contents of the quadword 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 element 0 of \( \text{load\_data} \),
- the contents of the byte in storage at address \( \text{EA}+1 \) are placed into byte element 14 of \( \text{load\_data} \), and so forth until
- the contents of the byte in storage at address \( \text{EA}+15 \) are placed into byte element 0 of \( \text{load\_data} \).

\( \text{load\_data} \) is placed into \( \text{VSR}[\text{XT}] \).

Special Registers Altered
None

Load VSX Vector Indexed X-form

\[ \text{lxvx} \ XT,\text{RA},\text{RB} \]

- if \( TX=0 \) & \( \text{MSR.VSX}=0 \) then \( \text{VSX.Unavailable}() \)
- if \( TX=1 \) & \( \text{MSR.VEC}=0 \) then \( \text{Vector.Unavailable}() \)

\[ \text{EA} \leftarrow ((\text{RA}=0) ? 0 : \text{GPR}[\text{RA}]) + \text{GPR}[\text{RB}] \]

\[ \text{VSR}[32\times TX+T] \leftarrow \text{MEM}(\text{EA},16) \]

Let \( XT \) be the value \( 32 \times TX + T \).

Let the effective address (EA) be the sum of the contents of \( \text{GPR}[\text{RA}] \), or 0 if \( \text{RA} \) is equal to 0, and the contents of \( \text{GPR}[\text{RB}] \).

When Big-Endian byte ordering is employed, the contents of the quadword 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 element 0 of \( \text{load\_data} \),
- the contents of the byte in storage at address \( \text{EA}+1 \) are placed into byte element 1 of \( \text{load\_data} \), and so forth until
- the contents of the byte in storage at address \( \text{EA}+15 \) are placed into byte element 15 of \( \text{load\_data} \).

When Little-Endian byte ordering is employed, the contents of the quadword 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 element 15 of \( \text{load\_data} \),
- the contents of the byte in storage at address \( \text{EA}+1 \) are placed into byte element 14 of \( \text{load\_data} \), and so forth until
- the contents of the byte in storage at address \( \text{EA}+15 \) are placed into byte element 0 of \( \text{load\_data} \).

\( \text{load\_data} \) is placed into \( \text{VSR}[\text{XT}] \).

Special Registers Altered:
None
Example:  

Loading data using Load VSX Vector Indexed

```c
char  W[16] = { 0xF0, 0xF1, 0xF2, 0xF3, 0xF4, 0xF5, 0xF6, 0xF7, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07 };  
short X[8] = { 0xF0F1, 0xF1F2, 0xF3F4, 0xF5F6, 0xF7F8, 0xE0E1, 0xE2E3, 0xE4E5 };  
float Y[4] = { 0xF0F1_F2F3, 0xF4F5_F6F7, 0xE0E1_E2E3, 0xE4E5_E6E7 };  
double Z[2] = { 0xF0F1_F2F3_F4F5_F6F7, 0xE0E1_E2E3_E4E5_E6E7 };  
```

Loading 16 bytes of data from Big-Endian storage in VSR[XT] using `lxvx`.

Big-endian storage image of W, X, Y, & Z:

```
addr(W+0x0000):  
F0 F1 F2 F3 F4 F5 F6 F7 E0 E1 E2 E3 E4 E5 E6 E7  
addr(W+0x0010):  
F0 F1 F2 F3 F4 F5 F6 F7 E0 E1 E2 E3 E4 E5 E6 E7  
addr(W+0x0020):  
F0 F1 F2 F3 F4 F5 F6 F7 E0 E1 E2 E3 E4 E5 E6 E7  
addr(W+0x0030):  
F0 F1 F2 F3 F4 F5 F6 F7 E0 E1 E2 E3 E4 E5 E6 E7  
```

# 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

```
lxvx   xW, r0, rPW  
lxvx   xX, r0, rPX  
lxvx   xY, r0, rPY  
lxvx   xZ, r0, rPZ  
```

Final state of VSRs W, X, Y, & Z:

```
VSR[W]:  
F0 F1 F2 F3 F4 F5 F6 F7 E0 E1 E2 E3 E4 E5 E6 E7  
VSR[X]:  
F0 F1 F2 F3 F4 F5 F6 E0 E1 E2 E3 E4 E5 E6 E7  
VSR[Y]:  
F0 F1 F2 F3 F4 F5 F6 F7 E0 E1 E2 E3 E4 E5 E6 E7  
VSR[Z]:  
F0 F1 F2 F3 F4 F5 F6 F7 E0 E1 E2 E3 E4 E5 E6 E7  
```

Little-endian storage image of W, X, Y, & Z:

```
addr(W+0x0000):  
F0 F1 F2 F3 F4 F5 F6 F7 E0 E1 E2 E3 E4 E5 E6 E7  
addr(W+0x0010):  
F1 F0 F2 F3 F4 F5 F6 F7 E1 E0 E2 E3 E4 E5 E6 E7  
addr(W+0x0020):  
F3 F2 F1 F0 F7 F6 F5 F4 E2 E1 E3 E4 E5 E6 E7  
addr(W+0x0030):  
F7 F6 F5 F4 F3 F2 F1 F0 E7 E6 E5 E4 E3 E2 E1 E0  
```

# 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

```
lxvx   xW, r0, rPW  
lxvx   xX, r0, rPX  
lxvx   xY, r0, rPY  
lxvx   xZ, r0, rPZ  
```

Final state of VSRs W, X, Y, & Z:

```
VSR[W]:  
E7 E6 E5 E4 E3 E2 E1 E0 F7 F6 F5 F4 F3 F2 F1 F0  
VSR[X]:  
E6 E7 E4 E5 E2 E3 E0 E1 F6 F7 F5 F4 F3 F2 F0 F1  
VSR[Y]:  
E4 E5 E6 E7 E0 E1 E2 E3 F4 F5 F6 F7 F0 F1 F2 F3  
VSR[Z]:  
E0 E1 E2 E3 E4 E5 E6 E7 F0 F1 F2 F3 F4 F5 F6 F7  
```
Load VSX Vector Doubleword & Splat Indexed X-form

lxvdsx XT,RA,RB

<table>
<thead>
<tr>
<th>31</th>
<th>8</th>
<th>32</th>
<th>24</th>
<th>16</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>R</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>332</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

if MSR.VSX=0 then VSXUnavailable()

EA ← ((RA=0) ? 0 : GPR[RA]) + GPR[RB]

load_data ← MEM(EA, 8)
VSR[32×TX+T].dword[0] ← load_data
VSR[32×TX+T].dword[1] ← load_data

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 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 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+7 are placed into byte element 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 element 7 of load_data,
- the contents of the byte in storage at address EA+1 are placed into byte element 6 of load_data, and so forth until
- the contents of the byte in storage at address EA+7 are placed into byte element 0 of load_data.

load_data is copied into each doubleword element of VSR[XT].

Special Registers Altered
None

VSR Data Layout for lxvdsx
tgt = VSR[XT]

<table>
<thead>
<tr>
<th>.dword[0]</th>
<th>.dword[1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 64 127</td>
<td>64 127</td>
</tr>
</tbody>
</table>
**Load VSX Vector Halfword*8 Indexed X-form**

The `lxvh8x` instruction is used to load 8 halfwords from memory into the VSX register `VSR[XT]`. The syntax is:

```
xvh8x XT, RA, RB
```

Where:
- `XT` is the target register.
- `RA` and `RB` are the source registers, with `RA` being the base register and `RB` being the index register.
- `T` is the index register offset.

### Example: Loading data using `Load VSX Vector Halfword*8 Indexed`

**Big-endian storage image of X**

<table>
<thead>
<tr>
<th>addr(X):</th>
<th>00 01 10 11 20 21 30 31 40 41 50 51 60 61 70 71</th>
</tr>
</thead>
<tbody>
<tr>
<td>0123456789AB</td>
<td>C D E F</td>
</tr>
</tbody>
</table>

**Little-endian storage image of X**

<table>
<thead>
<tr>
<th>addr(X):</th>
<th>01 00 11 10 21 20 31 30 41 40 51 50 61 60 71 70</th>
</tr>
</thead>
<tbody>
<tr>
<td>0123456789AB</td>
<td>C D E F</td>
</tr>
</tbody>
</table>

Loading a vector of 8 halfword elements from Big-Endian storage in `VSR[XT]` using `lxvh8x`, retaining left-to-right element ordering.

```c
short X[] = { 0x0001, 0x1011, 0x2021, 0x3031, 0x4041, 0x5051, 0x6061, 0x7071 };
```

**Loading a vector of 8 halfword elements from Little-Endian storage in `VSR[XT]` using `lxvh8x`, retaining left-to-right element ordering.**

```c
short X[] = { 0x7071, 0x6061, 0x5051, 0x4041, 0x3031, 0x2021, 0x1011, 0x0001 };
```

### Special Registers Altered:

- None

---

**Programming Note**

The instructions `lxvd2x`, `lxvw4x`, `lxvh8x`, `lxvb16x`, and `lxvx` exhibit identical behavior in Big-Endian mode.

---

**Chapter 7. Vector-Scalar Floating-Point Operations**
Load VSX Vector Word*4 Indexed X-form

lxvw4x XT,RA, RB

<table>
<thead>
<tr>
<th>31</th>
<th>T</th>
<th>RA</th>
<th>RB</th>
<th>780</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>11</td>
<td>16</td>
<td>21</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 byte element 0 of load_data,
- 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,
- 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

VSR Data Layout for lxvw4x

tgt = VSR[XT]

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>32</td>
<td>64</td>
<td>96</td>
</tr>
</tbody>
</table>

Programming Note
lxvd2x, lxvw4x, lxvh8x, lxvb16x, and lxvx exhibit identical behavior in Big-Endian mode.
**Load VSX Vector Word & Splat Indexed X-form**

\[ \text{lxvwsx} \quad \text{XT,RA,RB} \]

<table>
<thead>
<tr>
<th>31</th>
<th>T</th>
<th>RA</th>
<th>RB</th>
<th>364</th>
<th>7x</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>T</td>
<td>RA</td>
<td>RB</td>
<td>364</td>
<td>7x</td>
</tr>
</tbody>
</table>

if \( TX=0 \) & MSR.VSX=0 then VSX_Unavailable()
if \( TX=1 \) & MSR.VEC=0 then Vector_Unavailable()

\[ \text{EA} \leftarrow (\text{RA}=0 ? 0 : \text{GPR}[RA]) + \text{GPR}[RB] \]

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

\[ \text{do i = 0 to 3} \]

\[ \text{VSR}[32 \times TX + T].\text{word}[i] \leftarrow \text{load_data} \]

\[ \text{end} \]

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].

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

\[ \text{int } X = 0xF0F1_F2F3; \]

**Big-endian storage image of X**

\[
\begin{array}{cccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & A & B & C & D & E \\
F0 & F1 & F2 & F3 & 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 \\
\end{array}
\]

**Little-endian storage image of X**

\[
\begin{array}{cccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & A & B & C & D & E \\
F3 & F2 & F1 & F0 & 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 \\
\end{array}
\]

Loading scalar word data from Big-Endian storage in VSR[XT] using \text{l.vxwvx}:

\[
\text{# Assumptions} \\
\text{# GPR[PX] = address of X} \\
\text{lxvwsx } xx, r0, rPX}
\]

**Final state of VSR X**

\[
\begin{array}{cccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & A & B & C & D & E \\
F0 & F1 & F2 & F3 & F0 & F1 & F2 & F3 & F0 & F1 & F2 & F3 & F0 & F1 & F2 & F3 \\
\end{array}
\]

Loading scalar word data from Little-Endian storage in VSR[XT] using \text{l.vxwvx}:

\[
\text{# Assumptions} \\
\text{# GPR[PX] = address of X} \\
\text{lxvwsx } xx, r0, rPX}
\]

**Final state of VSR X**

\[
\begin{array}{cccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & A & B & C & D & E \\
F0 & F1 & F2 & F3 & F0 & F1 & F2 & F3 & F0 & F1 & F2 & F3 & F0 & F1 & F2 & F3 \\
\end{array}
\]
Store VSX Scalar Doubleword DS-form

\[
\text{stxsd } \text{VRS,DS(RA)}
\]

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>61</td>
<td>VRS</td>
</tr>
<tr>
<td>6</td>
<td>RA</td>
</tr>
<tr>
<td>10</td>
<td>DS</td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

Let \( X_S \) be the value \( \text{VRS} + 32 \).

Let \( E_A \) be the sum of the contents of \( \text{GPR}[\text{RA}] \), or 0 if \( \text{RA}=0 \), and the signed integer value \( \text{DS} \ll 2 \).

Let \( \text{store_data} \) be the contents of doubleword element 0 of \( \text{VSR}[X_S] \).

When Big-Endian byte ordering is employed, \( \text{store_data} \) is placed in the doubleword in storage at address \( E_A \) in such order that;

- byte 0 of \( \text{store_data} \) is placed into the byte in storage at address \( E_A \),
- byte 1 of \( \text{store_data} \) is placed into the byte in storage at address \( E_A+1 \), and so forth until
- byte 7 of \( \text{store_data} \) is placed into the byte in storage at address \( E_A+7 \).

When Little-Endian byte ordering is employed, \( \text{store_data} \) is placed in the doubleword in storage at address \( E_A \) in such order that;

- the contents of byte 7 of doubleword element 0 of \( \text{VSR}[\text{VRS}+32] \) are placed into the byte in storage at address \( E_A \),
- the contents of byte 6 of doubleword element 0 of \( \text{VSR}[\text{VRS}+32] \) are placed into the byte in storage at address \( E_A+1 \), and so forth until
- the contents of byte 0 of doubleword element 0 of \( \text{VSR}[\text{VRS}+32] \) are placed into the byte in storage at address \( E_A+7 \).

Special Registers Altered:
None

VSR Data Layout for stxsd

\[
\begin{array}{c|c|c}
0 & . \text{dword}[0] & \text{unused} \\
64 & 127 &
\end{array}
\]

Store VSX Scalar Doubleword Indexed X-form

\[
\text{stxsdx } X_S,\text{RA,RB}
\]

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RA</td>
</tr>
<tr>
<td>16</td>
<td>RB</td>
</tr>
<tr>
<td>21</td>
<td>716</td>
</tr>
</tbody>
</table>

Let \( X_S \) be the value \( 32 \times X_S + S \).

Let \( E_A \) be the sum of the contents of \( \text{GPR}[\text{RA}] \), or 0 if \( \text{RA} \) is equal to 0, and the contents of \( \text{GPR}[\text{RB}] \).

Let \( \text{store_data} \) be the contents of doubleword element 0 of \( \text{VSR}[X_S] \).

When Big-Endian byte ordering is employed, \( \text{store_data} \) is placed in the doubleword in storage at address \( E_A \) in such order that;

- byte 0 of \( \text{store_data} \) is placed into the byte in storage at address \( E_A \),
- byte 1 of \( \text{store_data} \) is placed into the byte in storage at address \( E_A+1 \), and so forth until
- byte 7 of \( \text{store_data} \) is placed into the byte in storage at address \( E_A+7 \).

When Little-Endian byte ordering is employed, \( \text{store_data} \) is placed in the doubleword in storage at address \( E_A \) in such order that;

- the contents of byte 7 of doubleword element 0 of \( \text{VSR}[\text{VRS}+32] \) are placed into the byte in storage at address \( E_A+7 \),
- the contents of byte 6 of doubleword element 0 of \( \text{VSR}[\text{VRS}+32] \) are placed into the byte in storage at address \( E_A+6 \), and so forth until
- the contents of byte 0 of doubleword element 0 of \( \text{VSR}[\text{VRS}+32] \) are placed into the byte in storage at address \( E_A \).

Special Registers Altered:
None

VSR Data Layout for stxsdx

\[
\begin{array}{c|c|c}
0 & . \text{dword}[0] & \text{unused} \\
64 & 127 &
\end{array}
\]
Store VSX Scalar as Integer Byte Indexed X-form

stxsibx XS,RA,RB

\[
\begin{array}{cccccccc}
31 & S & RA & RB & 909 & 0
\end{array}
\]

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

src = VSR[XS]

<table>
<thead>
<tr>
<th>unused</th>
<th>byte</th>
<th>unused</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>56</td>
<td>64</td>
</tr>
</tbody>
</table>

Store VSX Scalar as Integer Halfword Indexed X-form

stxsihx XS,RA,RB

\[
\begin{array}{cccccccc}
31 & S & RA & RB & 941 & 0
\end{array}
\]

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

src = VSR[XS]

<table>
<thead>
<tr>
<th>unused</th>
<th>hword[3]</th>
<th>unused</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>48</td>
<td>64</td>
</tr>
</tbody>
</table>
Store VSX Scalar as Integer Word Indexed X-form

\[
\text{stxsiwx} \quad \text{XS,RA,RB}
\]

| 31 | 140 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 31 |   S |   RA|   RB|   140|   SX |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

if MSR.VSX=0 then VSX_Unavailable();

\[
\text{EA} \leftarrow (\text{RA}=0) \lor (\text{GPR}[RA]) + \text{GPR}[RB]
\]

\[
\text{MEM(EA,4)} \leftarrow \text{VSR}[32\times\text{SX}+S].\text{word}[1]
\]

Let \( XS \) be the value \( 32 \times SX + S \).

Let \( EA \) be the sum of the contents of \( \text{GPR}[RA] \), or 0 if \( RA \) is equal to 0, and the contents of \( \text{GPR}[RB] \).

Let \( \text{store\_data} \) be the contents of word element 1 of \( \text{VSR}[XS] \).

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 \( EA \).

Special Registers Altered

None
Store VSX Scalar Single DS-form

\[
\text{stxssp VRS,DS(RA)}
\]

<table>
<thead>
<tr>
<th>61</th>
<th>VRS</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>3</td>
<td>30 31</td>
</tr>
</tbody>
</table>

if MSR.VEC=0 then Vector_Unavailable()

\[
\begin{align*}
\text{EA} & \leftarrow \begin{cases}
\text{RA}=0 & \Rightarrow \text{GPR}[\text{RA}] + \text{EXTS}([\text{DS}][0b00]) \\
0 & \Rightarrow \text{GPR}[\text{RA}] + \text{EXTS}([\text{DS}][0b00])
\end{cases} \\
\text{MEM}(\text{EA},4) & \leftarrow \text{ConvertSP64toSP(VSR[VRS+32].dword[0])}
\end{align*}
\]

Let \( XS \) be the value \( VRS + 32 \).

Let \( EA \) be the sum of the contents of \( \text{GPR}[\text{RA}] \), or 0 if \( \text{RA}=0 \), and the signed integer value \( \text{DS}[0b00] \).

Let \( \text{store\_data} \) be the double-precision floating-point value in doubleword element \( 0 \) of \( \text{VSR}[XS] \) converted to single-precision format.

When Big-Endian byte ordering is employed, \( \text{store\_data} \) is placed in the word in storage at address \( \text{EA} \) 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 \( \text{EA}+3 \).

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
Store VSX Scalar Single-Precision Indexed X-form

\[
\text{stxsspx } \text{XS,RA,RB}
\]

\[
\begin{array}{cccccc}
0 & 31 & S & RA & RB & 652 \\
6 & 11 & 16 & 21 & 31 \\
\end{array}
\]

\!
\begin{align*}
\text{if MSR.VSX=0 then VSX_Unavailable()} \\ 
\text{EA }\leftarrow (\text{if } \text{RA}=0 \text{ then } 0 \text{ else } \text{GPR}[\text{RA}]) + \text{GPR}[\text{RB}] \\
\text{MEN}[\text{EA},4] \leftarrow \text{ConvertSP64toSP(VSR[32\times\text{XS}].\text{dword}[0])} \\
\end{align*}

Let \( X_S \) be the value \( 32 \times X_S + S \).

Let \( EA \) be the sum of the contents of \( \text{GPR}[\text{RA}] \), or 0 if \( \text{RA} \) is equal to 0, and the contents of \( \text{GPR}[\text{RB}] \).

Let \( \text{store\_data} \) be the double-precision floating-point value in doubleword element 0 of \( \text{VSR}[X_S] \) 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 \( EA \).

Special Registers Altered

None
Store VSX Vector Byte*16 Indexed X-form

\[ \text{stxvb16x} \quad \text{XS,RA,RB} \]

<table>
<thead>
<tr>
<th>31</th>
<th>S</th>
<th>RA</th>
<th>RB</th>
<th>1004</th>
<th>Sx</th>
</tr>
</thead>
</table>

Let \( XS \) be the value \( 32 \times S + S \).

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,} \text{ and stxvx}

exhibit identical behavior in Big-Endian mode.

Example:
Storing data using Store VSX Vector Byte*16 Indexed

\[ \text{char X[16];} \]

\[ \text{VSR(X):} \]

\[ \begin{array}{cccccccc}
  0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
  A & B & C & D & E & F & 0 & 1 \\
  2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
  1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
\end{array} \]

Storing a vector of 16 byte elements from VSR[XS] into Big-Endian storage using \text{sxvb16x}, retaining left-to-right element ordering.

\[ \text{stxvb16x xX,r0,rPX} \]

Big-endian storage image of X

\[ \text{addr(X):} \]

\[ \begin{array}{cccccccc}
  0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
  A & B & C & D & E & F & 0 & 1 \\
  2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
  1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
\end{array} \]

Loading a vector of 16 byte elements from Little-Endian storage in VSR[XT] using \text{lxvb16x}, retaining left-to-right element ordering.

\[ \text{stxvb16x xX,r0,rPX} \]

Little-endian storage image of X

\[ \text{addr(X):} \]

\[ \begin{array}{cccccccc}
  0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
  A & B & C & D & E & F & 0 & 1 \\
  2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
  1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
\end{array} \]
### Store VSX Vector Doubleword*2 Indexed X-form

**stxvd2x**

```
X, RA, RB
```

<table>
<thead>
<tr>
<th>S</th>
<th>RA</th>
<th>RB</th>
<th>972</th>
<th>X</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()
- \( EA \leftarrow ((RA=0) \ ? \ 0 : \text{GPR}[RA]) + \text{GPR}[RB] \)
- \( \text{MEM}(EA,8) \leftarrow \text{VSR}[32 \times SX+S].dword[0] \)
- \( \text{MEM}(EA+8,8) \leftarrow \text{VSR}[32 \times SX+S].dword[1] \)

Let \( XS \) be the value \( 32 \times SX + S \).

Let \( EA \) be the sum of the contents of \( \text{GPR}[RA] \), or 0 if \( RA \) is equal to 0, and the contents of \( \text{GPR}[RB] \).

For each integer value \( i \) from 0 to 1, do the following.
- Let \( \text{store data} \) be the contents of doubleword element \( i \) of \( \text{VSR}[XS] \).

When Big-Endian byte ordering is employed, \( \text{store data} \) is placed in the doubleword in storage at address \( EA+8 \times i \) in such order that;
- byte 0 of \( \text{store data} \) is placed into the byte in storage at address \( EA+8 \times i \),
- byte 1 of \( \text{store data} \) is placed into the byte in storage at address \( EA+8 \times i+1 \), and so forth until
- byte 7 of \( \text{store data} \) is placed into the byte in storage at address \( EA+8 \times i+7 \).

When Little-Endian byte ordering is employed, \( \text{store data} \) is placed in the doubleword in storage at address \( EA+8 \times i \) in such order that;
- byte 0 of \( \text{store data} \) is placed into the byte in storage at address \( EA+8 \times i+7 \),
- byte 1 of \( \text{store data} \) is placed into the byte in storage at address \( EA+8 \times i+6 \), and so forth until
- byte 7 of \( \text{store data} \) is placed into the byte in storage at address \( EA+8 \times i \).

### Special Registers Altered

None

### VSR Data Layout for stxvd2x

```
src = VSR[X]
```

<table>
<thead>
<tr>
<th>.dword[0]</th>
<th>.dword[1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>64</td>
</tr>
</tbody>
</table>
Store VSX Vector Halfword*8 Indexed X-form

stxvh8x XS,RA,RB

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 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+2xi 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+2xi, 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+2xi+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+2xi 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+2xi, 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+2xi+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):

<table>
<thead>
<tr>
<th>00</th>
<th>01</th>
<th>10</th>
<th>11</th>
<th>20</th>
<th>21</th>
<th>30</th>
<th>31</th>
<th>40</th>
<th>41</th>
<th>50</th>
<th>51</th>
<th>60</th>
<th>61</th>
<th>70</th>
<th>71</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>00</td>
<td>10</td>
<td>11</td>
<td>21</td>
<td>20</td>
<td>31</td>
<td>30</td>
<td>41</td>
<td>40</td>
<td>51</td>
<td>50</td>
<td>61</td>
<td>60</td>
<td>71</td>
<td>70</td>
</tr>
</tbody>
</table>

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):

<table>
<thead>
<tr>
<th>00</th>
<th>01</th>
<th>10</th>
<th>11</th>
<th>20</th>
<th>21</th>
<th>30</th>
<th>31</th>
<th>40</th>
<th>41</th>
<th>50</th>
<th>51</th>
<th>60</th>
<th>61</th>
<th>70</th>
<th>71</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>00</td>
<td>10</td>
<td>11</td>
<td>21</td>
<td>20</td>
<td>31</td>
<td>30</td>
<td>41</td>
<td>40</td>
<td>51</td>
<td>50</td>
<td>61</td>
<td>60</td>
<td>71</td>
<td>70</td>
</tr>
</tbody>
</table>

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 Word*4 Indexed X-form

stxvw4x  XS,RA,RB

<table>
<thead>
<tr>
<th>31</th>
<th>S</th>
<th>RA</th>
<th>RB</th>
<th>908</th>
<th>3x</th>
</tr>
</thead>
</table>

if MSR.VSX=0 then VSX_Unavailable()

EA ← ((RA=0) ? 0 : GPR[RA]) + GPR[RB]

MEM(EA,4) ← VSR[32×SX+S].word[0]
MEM(EA+4,4) ← VSR[32×SX+S].word[1]
MEM(EA+8,4) ← VSR[32×SX+S].word[2]
MEM(EA+12,4) ← VSR[32×SX+S].word[3]

Let XS be the value 32×S+X.

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.

Let store_data be the contents of word element i of VSR[XS].

When Big-Endian byte ordering is employed, store_data is placed in the word in storage at address EA+i×4 in such order that;

- byte 0 of store_data is placed into the byte in storage at address EA+i×4,
- byte 1 of store_data is placed into the byte in storage at address EA+i×4+1, and so forth until
- byte 3 of store_data is placed into the byte in storage at address EA+i×4+3.

When Little-Endian byte ordering is employed, store_data is placed in the word in storage at address EA+i×4 in such order that;

- byte 0 of store_data is placed into the byte in storage at address EA+i×4+3,
- byte 1 of store_data is placed into the byte in storage at address EA+i×4+2, and so forth until
- byte 3 of store_data is placed into the byte in storage at address EA+i×4.

Special Registers Altered

None

VSR Data Layout for stxvw4x

src = VSR[XS]

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>32</td>
<td>64</td>
<td>96</td>
</tr>
</tbody>
</table>

Programming Note

stxvd2x, stxvw4x, stxvh8x, stxvb16x, and stxvx exhibit identical behavior in Big-Endian mode.
Store VSX Vector DQ-form

\textbf{stxv} \hspace{1em} X5, DQ(RA)

<table>
<thead>
<tr>
<th>61</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>SX</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

\textbf{Special Registers Altered}

\textbf{None}

Store VSX Vector with Length X-form

\textbf{stxvl} \hspace{1em} X5, RA, RB

<table>
<thead>
<tr>
<th>31</th>
<th>4</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>397</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

\textbf{Special Registers Altered}

\textbf{None}

Let \( XS \) be the value \( 32 \times S + S \).

Let \( EA \) be the sum of the contents of GPR[RA], or 0 if RA=0, and the signed integer value DQ<<4.

Let \( \text{store\_data} \) be the contents of VSR[XS].

When Big-Endian byte ordering is employed, \( \text{store\_data} \) is placed into the quadword 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 15 of \( \text{store\_data} \) is placed into the byte in storage at address \( EA+15 \).

When Little-Endian byte ordering is employed, \( \text{store\_data} \) is placed into the quadword in storage at address \( EA \) in such an order that;

- byte 15 of \( \text{store\_data} \) is placed into the byte in storage at address \( EA \),
- byte 14 of \( \text{store\_data} \) is placed into the byte in storage at address \( EA+1 \), and so forth until
- byte 0 of \( \text{store\_data} \) is placed into the byte in storage at address \( EA+15 \).

Let \( XS \) be the value \( 32 \times S + S \).

Let the effective address \( EA \) be the contents of GPR[RA], or 0 if RA=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, when Big-Endian byte-ordering is employed, do the following.

- If \( nb < 16 \), 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 \( EA \).

Otherwise, when Little-Endian byte ordering is employed, do the following.

- If \( nb < 16 \), the contents of the rightmost \( nb \) bytes of VSR[XS] are placed in storage starting at address \( EA \) in byte-reversed order.

Otherwise, the contents of VSR[XS] are placed into the quadword in storage at address \( EA \) in byte-reversed order.

If the contents of bits 8:63 of GPR[RB] are not equal to 0, the results are boundedly undefined.

\textbf{Special Registers Altered}

\textbf{None}
Example: Storing less than 16-byte data from VSR using stxvl

```c
char S[14] = "This is a TEST";
short X[6] = { 0x0E01, 0x0E03, 0x0E45, 0x0E67, 0xE8E9, 0xEAE8 };
```

Storing less than 16-byte data in VSR[XS] into Big-Endian storage using `stxvl`.

```
char        S[14] = "This is a TEST";
short       X[6]  = { 0xE0E1, 0xE2E3, 0xE4E5, 0xE6E7, 0xE8E9, 0xEAEB };
binary80    Z     = 0xF0F1F2F3F4F5F6F7F8F9
```

# 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

<table>
<thead>
<tr>
<th>VSR[S]</th>
<th>VSR[X]</th>
<th>VSR[Z]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

VSR register image of S, X, & Z

<table>
<thead>
<tr>
<th>VSR[S]:</th>
<th>VSR[X]:</th>
<th>VSR[Z]:</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;T&quot; &quot;h&quot; &quot;i&quot; &quot;s&quot; &quot; T&quot;</td>
<td>E0 E1 E2 E3 E4 E5 E6 E7 E8 E9 EA EB</td>
<td>F0 F1 F2 F3 F4 F5 F6 F7 F8 F9</td>
</tr>
</tbody>
</table>

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
stxvl  xS,rPS,rLS
stxvl  xX,rPX,rLX
stxvl  xZ,rPZ,rLZ

Final state of Big-Endian storage image of S, X, & Z

<table>
<thead>
<tr>
<th>addr(S)+0x0000:</th>
<th>addr(S)+0x0010:</th>
<th>addr(S)+0x0020:</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;T&quot; &quot;h&quot; &quot;i&quot; &quot;s&quot; &quot; T&quot;</td>
<td>E0 E1 E2 E3 E4 E5 E6 E7 E8 E9 EA EB</td>
<td>F0 F1 F2 F3 F4 F5 F6 F7 F8 F9</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>VSR[S]:</th>
<th>VSR[X]:</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;T&quot; &quot;h&quot; &quot;S&quot; &quot;T&quot; &quot;a&quot; &quot;a&quot;</td>
<td>E0 E1 E2 E3 E4 E5 E6 E7 E8 E9 EA EB</td>
</tr>
</tbody>
</table>

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
stxvl  xS,rPS,rLS
stxvl  xX,rPX,rLX
stxvl  xZ,rPZ,rLZ

Final state of Little-Endian storage image of S, X, & Z

<table>
<thead>
<tr>
<th>addr(S)+0x0000:</th>
<th>addr(S)+0x0010:</th>
<th>addr(S)+0x0020:</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;T&quot; &quot;h&quot; &quot;S&quot; &quot;T&quot; &quot;a&quot; &quot;a&quot;</td>
<td>E0 E1 E2 E3 E4 E5 E6 E7 E8 E9 EA EB</td>
<td>F0 F1 F2 F3</td>
</tr>
</tbody>
</table>

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
stxvl  xS,rPS,rLS
stxvl  xX,rPX,rLX
stxvl  xZ,rPZ,rLZ

Final state of Little-Endian storage image of S, X, & Z

<table>
<thead>
<tr>
<th>addr(S)+0x0000:</th>
<th>addr(S)+0x0010:</th>
<th>addr(S)+0x0020:</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;T&quot; &quot;h&quot; &quot;S&quot; &quot;T&quot; &quot;a&quot; &quot;a&quot;</td>
<td>E0 E1 E2 E3 E4 E5 E6 E7 E8 E9 EA EB</td>
<td>F0 F1 F2 F3</td>
</tr>
</tbody>
</table>

Store VSX Vector Left-justified with Length X-form

\[
stxvll \quad XS,RA,RB
\]

### Table

<table>
<thead>
<tr>
<th>S</th>
<th>RA</th>
<th>RB</th>
<th>429</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>0</td>
<td>11</td>
<td>10</td>
</tr>
</tbody>
</table>

If \( SX=0 \) & MSR.VSX=0 then VSX_Unavailable()
If \( SX=1 \) & MSR.VEC=0 then Vector_Unavailable()

\[
EA \leftarrow (RA=0) \oplus 0 : \text{GPR}[RA]
\]

\[
\text{nb} \leftarrow \text{EXTZ(GPR}[RB].\text{bit}[0:7])
\]

If \( \text{nb}>16 \) then \( \text{nb} \leftarrow 16 \)

If \( \text{nb}>0 \) then do
\[
\text{MEM(EA+i,1)} \leftarrow \text{VSR}[32\times SX+S].\text{byte}[i]
\]
end

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 \( \text{nb} \) be the unsigned integer value in bits 0:7 of GPR[RB].

If \( \text{nb} \) is equal to 0, the storage access is not performed.

Otherwise, do the following.

If \( \text{nb} \) less than 16, the contents of the leftmost \( \text{nb} \) 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

\[
\text{decimal } X = +1234567890123456789; \\
\text{decimal } Y = -123456; \\
\text{decimal } Z = +1004966723510220;
\]

Storing less than 16-byte data, left-justified in VSR[XS], into storage using \( \text{stxvll} \).

# 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

VSRs X, Y, & Z

Initial state of Big-endian & Little-Endian storage image of X, Y, & Z

<p>| | | | | | | | | | | | |</p>
<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>
</tr>
</thead>
<tbody>
<tr>
<td>VSR[X]:</td>
<td>01 34 67 78 90 12 34 56 78 9C</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VSR[Y]:</td>
<td>01 23 45 6D</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VSR[Z]:</td>
<td>01 00 49 66 72 35 10 22 0C</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Final state of Big-endian & Little-Endian storage image of X, Y, & Z

<p>| | | | | | | | | | | | |</p>
<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>
</tr>
</thead>
<tbody>
<tr>
<td>X+0x0000:</td>
<td>12 34 56 78 90 12 34 56 78 9C</td>
<td>01 23 45 6D</td>
<td>01 00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X+0x0010:</td>
<td>49 66 72 35 10 22 0C</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Store VSX Vector Indexed X-form**

```
stxvx   XS,RA,RB
```

<table>
<thead>
<tr>
<th>0</th>
<th>31</th>
<th>6</th>
<th>S</th>
<th>11</th>
<th>RA</th>
<th>16</th>
<th>RB</th>
<th>21</th>
<th>396</th>
<th>SX</th>
</tr>
</thead>
</table>

If $S = 0$ & MSR.VSX=0 then VSX_Unavailable();
If $S = 1$ & MSR.VEC=0 then Vector_Unavailable();

$$EA \leftarrow ((RA=0) \ ? \ 0 : \text{GPR}[RA]) + \text{GPR}[RB]$$

$$\text{MEM}(EA,16) \leftarrow \text{VSR}[32\times SX + S]$$

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].

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**

$\text{stxvd2x}, \text{stxvw4x}, \text{stxvh8x}, \text{stxvb16x}$, and $\text{stxvx}$ exhibit identical behavior in Big-Endian mode.
Example: Storing data using Store VSX Vector Indexed

char W[16] = { 0xF0, 0xF1, 0xF2, 0xF3, 0xF4, 0xF5, 0xF6, 0xF7, 0xE0, 0xE1, 0xE2, 0xE3, 0xE4, 0xE5, 0xE6, 0xE7 };  
short X[8] = { 0xF0F1, 0xF1F2, 0xF2F3, 0xF3F4, 0xF4F5, 0xF5F6, 0xF6F7, 0xE0E1 };  
float Y[4] = { 0xF0F1_F2F3_F4F5_F6F7, 0xE0E1_E2E3_E4E5_E6E7 };  
double Z[2] = { 0xF0F1_F2F3_F4F5_F6F7_F8F9_F0F1, 0xE0E1_E2E3_E4E5_E6E7_E8E9_E0F0 };

Storing 16 bytes of data into Big-Endian storage from VSR[X logically] using stvx:

VSR[W]:
F0 F1 F2 F3 F4 F5 F6 F7 E0 E1 E2 E3 E4 E5 E6 E7
VSR[X]:
F0 F1 F2 F3 F4 F5 F6 F7 E0 E1 E2 E3 E4 E5 E6 E7
VSR[Y]:
F0 F1 F2 F3 F4 F5 F6 F7 E0 E1 E2 E3 E4 E5 E6 E7
VSR[Z]:
F0 F1 F2 F3 F4 F5 F6 F7 E0 E1 E2 E3 E4 E5 E6 E7

Big-endian storage image of W, X, Y, & Z

addr(W+0x0000):
F0 F1 F2 F3 F4 F5 F6 F7 E0 E1 E2 E3 E4 E5 E6 E7
addr(W+0x0010):
F0 F1 F2 F3 F4 F5 F6 F7 E0 E1 E2 E3 E4 E5 E6 E7
addr(W+0x0020):
F0 F1 F2 F3 F4 F5 F6 F7 E0 E1 E2 E3 E4 E5 E6 E7
addr(W+0x0030):
F0 F1 F2 F3 F4 F5 F6 F7 E0 E1 E2 E3 E4 E5 E6 E7

# 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

stvx xW,r0,rPW
stvx xX,r0,rPX
stvx xY,r0,rPY
stvx xZ,r0,rPZ

Storing 16 bytes of data into Little-Endian storage from VSR[X logically] using stvx:

VSR[W]:
E7 E6 E5 E4 E3 E2 E1 E0 F7 F6 F5 F4 F3 F2 F1 F0
VSR[X]:
E6 E7 E4 E5 E2 E3 E0 E1 F6 F7 F4 F5 F2 F3 F0 F1
VSR[Y]:
E4 E5 E6 E7 E0 E1 E2 E3 F4 F5 F6 F7 F0 F1 F2 F3
VSR[Z]:
E0 E1 E2 E3 E4 E5 E6 E7 F0 F1 F2 F3 F4 F5 F6 F7

Little-endian storage image of W, X, Y, & Z

addr(W+0x0000):
F0 F1 F2 F3 F4 F5 F6 F7 E0 E1 E2 E3 E4 E5 E6 E7
addr(W+0x0010):
F1 F0 F3 F2 F5 F4 F7 F6 E0 E1 E2 E3 E4 E5 E6 E7
addr(W+0x0020):
F2 F1 F0 F3 F4 F5 F6 F7 E0 E1 E2 E3 E4 E5 E6 E7
addr(W+0x0030):
F3 F2 F1 F0 F7 F6 F5 F4 E0 E1 E2 E3 E4 E5 E6 E7

# 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

stvx xW,r0,rPW
stvx xX,r0,rPX
stvx xY,r0,rPY
stvx xZ,r0,rPZ
VSX Scalar Absolute Double-Precision XX2-form

-xsabsdp XT,XB

Let XT be the value \( 32 \times TX + T \).
Let XB be the value \( 32 \times BX + B \).

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 undefined.

Special Registers Altered
None

VSR Data Layout for xsabsdp

<table>
<thead>
<tr>
<th>src = VSR[XB]</th>
<th>tgt = VSR[XT]</th>
</tr>
</thead>
<tbody>
<tr>
<td>DP</td>
<td>unused</td>
</tr>
<tr>
<td>0</td>
<td>64</td>
</tr>
</tbody>
</table>

Programming Note
This instruction can be used to operate on a single-precision source operand.

VSX Scalar Absolute Quad-Precision X-form

-xsabsqp VRT,VRB

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

VSR Data Layout for xsabsqp

<table>
<thead>
<tr>
<th>VSR[XB]</th>
<th>VSR[XT]</th>
</tr>
</thead>
<tbody>
<tr>
<td>src</td>
<td>tgt</td>
</tr>
</tbody>
</table>
**VSX Scalar Add Double-Precision XX3-form**

{\texttt{xsadddp \ XT,XA,XB}}

<table>
<thead>
<tr>
<th></th>
<th>60</th>
<th>T</th>
<th>11</th>
<th>A</th>
<th>16</th>
<th>B</th>
<th>32</th>
<th>0B</th>
</tr>
</thead>
<tbody>
<tr>
<td>XT</td>
<td>← TX</td>
<td></td>
<td>T</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XA</td>
<td>← AX</td>
<td></td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XB</td>
<td>← BX</td>
<td></td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\(\text{reset}_x\text{flags}()\)

\(\text{src1} \leftarrow \text{VSR[XA]}[0:63]\)

\(\text{src2} \leftarrow \text{VSR[XB]}[0:63]\)

\(v[0:inf] \leftarrow \text{AddDP(src1, src2)}\)

\(\text{result}[0:63] \leftarrow \text{RoundToDP(RN,v)}\)

if \(v\text{xsnan\_flag}\) then Set\(FX(VXSNAN)\)

if \(v\text{xisi\_flag}\) then Set\(FX(VXISI)\)

if \(o\_flag\) then Set\(FX(OX)\)

if \(u\_flag\) then Set\(FX(UX)\)

if \(x\_flag\) then Set\(FX(XX)\)

\(\text{vex\_flag} \leftarrow \text{VE} \& (v\text{xsnan\_flag} \mid v\text{xisi\_flag})\)

if \(\neg\text{vex\_flag}\) then do

\(\text{VSR}[\text{XT}] \leftarrow \text{result} || 0xUUUU_UUUU_UUUU_UUUU\)

\(\text{FPRF} \leftarrow \text{ClassSP(result)}\)

\(\text{FR} \leftarrow \text{inc\_flag}\)

\(\text{FI} \leftarrow \text{xx\_flag}\)

end

else do

\(\text{FR} \leftarrow 0b0\)

\(\text{FI} \leftarrow 0b0\)

end

Let \(\text{XT}\) be the value \(32\times TX + T\).

Let \(\text{XA}\) be the value \(32\times AX + A\).

Let \(\text{XB}\) be the value \(32\times BX + B\).

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 undefined.

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 51, “VSX Scalar Floating-Point Final Result,” on page 516.

**Special Registers Altered**

\[
\begin{array}{cccccccc}
\text{FPRF} & \text{FR} & \text{FI} & \text{FX} & \text{OX} & \text{UX} & \text{XX} \\
\text{VXSNAN} & \text{VXISI} \\
\end{array}
\]

**VSR Data Layout for xsadddp**

\(\text{src1} = \text{VSR[XA]}\)

\[
\begin{array}{cccccccc}
\text{DP} & \text{unused} \\
\end{array}
\]

\(\text{src2} = \text{VSR[XB]}\)

\[
\begin{array}{cccccccc}
\text{DP} & \text{unused} \\
\end{array}
\]

\(\text{tgt} = \text{VSR[XT]}\)

\[
\begin{array}{cccccccc}
\text{DP} & \text{undefined} \\
\end{array}
\]

**VSX Data Layout for xsadddp**

Let \(\text{src1}\) be the double-precision floating-point value in doubleword element 0 of VSR[XA].

Let \(\text{src2}\) be the double-precision floating-point value in doubleword element 0 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 49, “Actions for xsadddp,” on page 514.

The intermediate result is rounded to double-precision using the rounding mode specified by RN.

See Table 50, “Scalar Floating-Point Intermediate Result Handling,” on page 515.

---

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).
- **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.
  - The intermediate result having unbounded significand precision and unbounded exponent range.

**Table 49. Actions for xsadddp**

<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 ← dQNaN</td>
<td>vxsi_flag ← 1</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(src1,src2)</td>
<td>v ← src1</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>-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) vxsnan_flag ← 1</td>
</tr>
<tr>
<td>+Zero</td>
<td>v ← -Infinity</td>
<td>v ← src2</td>
<td>v ← +Zero</td>
<td>v ← src2</td>
<td>v ← +Infinity</td>
<td>v ← src2</td>
<td>v ← +Infinity</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 ← +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>+Infinity</td>
<td>v ← dQNaN 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 ← +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) vxsnan_flag ← 1</td>
<td>v ← Q(src1) vxsnan_flag ← 1</td>
<td>v ← Q(src1) vxsnan_flag ← 1</td>
<td>v ← Q(src1) vxsnan_flag ← 1</td>
<td>v ← Q(src1) vxsnan_flag ← 1</td>
<td>v ← Q(src1) vxsnan_flag ← 1</td>
<td>v ← Q(src1) vxsnan_flag ← 1</td>
<td>v ← Q(src1) vxsnan_flag ← 1</td>
</tr>
</tbody>
</table>
### Table 50. 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 NaNv</td>
<td>Special</td>
<td>r ← v</td>
</tr>
<tr>
<td>v = -Infinity</td>
<td>Special</td>
<td>r ← v</td>
</tr>
<tr>
<td>-Infinity &lt; v &lt; (-Nmax + 1ulp)</td>
<td>Overflow</td>
<td>Q ← ind(v)</td>
</tr>
<tr>
<td>-Nmax &lt; v &lt; -Nmax</td>
<td>Normal</td>
<td>r ← -Nmax</td>
</tr>
<tr>
<td>-Nmax &lt; v &lt; -Zero</td>
<td>Normal</td>
<td>r ← -Nmax</td>
</tr>
<tr>
<td>v = -Zero</td>
<td>Special</td>
<td>r ← -Nmax</td>
</tr>
<tr>
<td>v = Rezd</td>
<td>Special</td>
<td>r ← -Nmax</td>
</tr>
<tr>
<td>-Nmax &lt; v &lt; -Nmax</td>
<td>Normal</td>
<td>r ← -Nmax</td>
</tr>
<tr>
<td>-Nmax &lt; v &lt; -Zero</td>
<td>Normal</td>
<td>r ← -Nmax</td>
</tr>
<tr>
<td>v = Rezd</td>
<td>Special</td>
<td>r ← -Nmax</td>
</tr>
<tr>
<td>v = -Zero</td>
<td>Special</td>
<td>r ← -Nmax</td>
</tr>
<tr>
<td>+Zero &lt; v &lt; +Nmax</td>
<td>Tiny</td>
<td>Q ← ind(v)</td>
</tr>
<tr>
<td>+Nmax &lt; v &lt; +Infinity</td>
<td>Normal</td>
<td>r ← +Infinity</td>
</tr>
<tr>
<td>+Nmax &lt; v &lt; +Infinity</td>
<td>Normal</td>
<td>r ← +Infinity</td>
</tr>
<tr>
<td>v = +Infinity</td>
<td>Special</td>
<td>r ← +Infinity</td>
</tr>
</tbody>
</table>

#### Explanation:

- This situation cannot occur.
- The precise intermediate result defined in the instruction having unbounded range and precision.
- The significand of v is shifted right by the amount of the difference between the target rounding precision Em,n and the unbiased exponent of v. The unbiased exponent of the denormalized value is Em,r. The significand of the denormalized value has unbounded significand precision.

- **Rezd**
  - Exact-zero-difference result. Applies only to addition 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 controlled by the setting of the rounding mode in RN, even when the rounding mode is overridden to Round to Odd.

- **rden(x)**
  - The significand of x is rounded to the target rounding precision according to the rounding mode specified in FPCR.RN. Exponent range of the rounded result is unbounded. See Section 7.3.2.6.

- **Nmax**
  - Largest (in magnitude) representable normalized number in the target rounding precision format.

- **Nm n**
  - Smallest (in magnitude) representable normalized number in the target rounding precision format.

- **ulp**
  - Least significant bit in the target precision format’s significand (Unit in the Last Position).
<table>
<thead>
<tr>
<th>Case</th>
<th>FPSCR.VE</th>
<th>FPSCR.VE</th>
<th>FPSCR.VE</th>
<th>FPSCR.ZE</th>
<th>FPSCR.ZE</th>
<th>Vxsnan_{r}</th>
<th>Vxsi_{r}</th>
<th>Vxdi_{r}</th>
<th>Vxid_{r}</th>
<th>Vxdz_{r}</th>
<th>Vxsqrt_{r}</th>
<th>zx_{r}</th>
<th>Returned Results and Status Setting</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>T(r), class_bfp(r), fi(0), fr(0), fx(ZX)</td>
</tr>
<tr>
<td>0</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>T(r), class_bfp(r), fi(0), fr(0), fx(VXSNAN)</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>T(r), class_bfp(r), fi(0), fr(0), fx(VXISI)</td>
</tr>
<tr>
<td>1</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>T(r), class_bfp(r), fi(0), fr(0), fx(VXIMZ)</td>
</tr>
<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>
<td>T(r), class_bfp(r), fi(0), fr(0), fx(VXSNAN), fx(VXIMZ)</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>
<td>-</td>
<td>T(r), class_bfp(r), fi(0), fr(0), fx(VXZDZ)</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>
<td>-</td>
<td>T(r), class_bfp(r), fi(0), fr(0), fx(VXISI)</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>
<td>-</td>
<td>T(r), class_bfp(r), fi(0), fr(0), fx(VXIMZ)</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>
<td>-</td>
<td>T(r), class_bfp(r), fi(0), fr(0), fx(VXSNAN), fx(VXISI)</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
    \[ VSR[XT].dword[0] = bfp\_CONVERT\_TO\_BFP64(r) \]
    \[ VSR[XT].dword[1] = 0xUUUU_UUUU_UUUU_UUUU \]
  - For scalar quad-precision results
    \[ VSR[VRT + 32] = bfp\_CONVERT\_TO\_BFP128(r) \]
- \( \text{class\_bfp}(x) \) 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)
- \( f_x(x) \) FPSCR.FX is set to 1 if \( \text{FPSCR}.x=0 \).
  - FPSCR.x is set to 1.
- \( f_i(x) \) FPSCR.FI is set to the value \( x \).
- \( f_r(x) \) FPSCR.FR is set to the value \( x \).
- \( \beta \) Wrap adjust
  - \( \beta = 2^{192} \) (single-precision)
  - \( \beta = 2^{1536} \) (double-precision)
  - \( \beta = 2^{24576} \) (quad-precision)
- See Table 7.4.3.2, “Action for OE=1,” on page 404 for trap-enabled Overflow exceptions.
- See Table 7.4.4.2, “Action for UE=1,” on page 409 for trap-enabled Underflow exceptions.
- \( q \) The value defined in Table 50, “Scalar Floating-Point Intermediate Result Handling,” on page 515, significand rounded to the target rounding precision, unbounded exponent range.
- \( r \) The value defined in Table 50, “Scalar Floating-Point Intermediate Result Handling,” on page 515, 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 MSR.FE0 and MSR.FE1 are set to any mode other than the ignore-exception mode.

Table 51. VSX Scalar Floating-Point Final Result

516 Power ISA™ I
### Table 51. VSX Scalar Floating-Point Final Result (Continued)

<table>
<thead>
<tr>
<th>Case</th>
<th>Returned Results and Status Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal</td>
<td></td>
</tr>
<tr>
<td>Overflow</td>
<td></td>
</tr>
<tr>
<td>Tiny</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
- For scalar quad-precision results

**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. FPSCR.x is set to 1.

**fi[x]**
FPSCR.FI is set to the value i.

**fr[x]**
FPSCR.FR is set to the value i.

**β**
Wrap adjust
- \( β = \frac{1}{2} \) (single-precision)
- \( β = \frac{1}{2} \) (double-precision)
- \( β = \frac{1}{2} \) (quad-precision)

See Table 7.4.3.2, “Action for OE=1,” on page 404 for trap-enabled Overflow exceptions.

See Table 7.4.4.2, “Action for UE=1,” on page 409 for trap-enabled Underflow exceptions.

**q**
The value defined in Table 50, “Scalar Floating-Point Intermediate Result Handling,” on page 515, significand rounded to the target rounding precision, unbounded exponent range.

**r**
The value defined in Table 50, “Scalar Floating-Point Intermediate Result Handling,” on page 515, 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.
### VSX Scalar Add Single-Precision XX3-form

<table>
<thead>
<tr>
<th>xsaddsp</th>
<th>XT,XA,XB</th>
</tr>
</thead>
<tbody>
<tr>
<td>60</td>
<td>6</td>
</tr>
<tr>
<td>T</td>
<td>A</td>
</tr>
<tr>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>64</td>
</tr>
</tbody>
</table>

reset_xflags()

\[
\begin{align*}
src1 & \leftarrow \text{VSR}[32 \times AX + A].dword[0] \\
src2 & \leftarrow \text{VSR}[32 \times BX + B].dword[0] \\
v & \leftarrow \text{AddDP}(src1, src2) \\
result & \leftarrow \text{RoundToSP}(RN, v) \\
\end{align*}
\]

if(vxsnan_flag) then SetFX(VXSNAN)
if(vxisi_flag) then SetFX(VXISI)
if(xx_flag) then SetFX(XX)
if(ux_flag) then SetFX(UX)
if(os_flag) then SetFX(OX)

\[
\begin{align*}
vex_flag & \leftarrow \text{VE} \& (\text{vxsnan_flag} \mid \text{vxisi_flag}) \\
\end{align*}
\]

if( ~vex_flag ) then do
\[
\begin{align*}
\text{VSR}[32 \times TX + T].dword[0] & \leftarrow \text{ConvertSPtoDP}(\text{result}) \\
\text{VSR}[32 \times TX + T].dword[1] & \leftarrow 0xUUUU_UUUU_UUUU_UUUU \\
\text{FPRF} & \leftarrow \text{ClassSP}(\text{result}) \\
\text{FR} & \leftarrow \text{inc_flag} \\
\text{FI} & \leftarrow xx_flag \\
\end{align*}
\]

end
else do
\[
\begin{align*}
\text{FR} & \leftarrow 0b0 \\
\text{FI} & \leftarrow 0b0 \\
\end{align*}
\]

end

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 result is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are undefined.

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 51, “VSX Scalar Floating-Point Final Result,” on page 516.

#### Special Registers Altered

<table>
<thead>
<tr>
<th>FPRF</th>
<th>FR</th>
<th>FI</th>
<th>FX</th>
<th>OX</th>
<th>UX</th>
<th>XX</th>
</tr>
</thead>
<tbody>
<tr>
<td>VXSNAN</td>
<td>VXISI</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### VSR Data Layout for xsaddsp

<table>
<thead>
<tr>
<th>src1 = VSR[AX]</th>
</tr>
</thead>
<tbody>
<tr>
<td>DP</td>
</tr>
<tr>
<td>src2 = VSR[XB]</td>
</tr>
<tr>
<td>---------------</td>
</tr>
<tr>
<td>DP</td>
</tr>
<tr>
<td>tgt = VSR[XT]</td>
</tr>
<tr>
<td>---------------</td>
</tr>
<tr>
<td>DP</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.

---

518  Power ISA™ I
## Chapter 7. Vector-Scalar Floating-Point Operations

### Table 52. 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>src1</td>
<td>src2</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← dQNaN</td>
<td>v ← +Infinity</td>
<td>v ← src2</td>
<td>v ← dQNaN</td>
</tr>
<tr>
<td>src1</td>
<td>src2</td>
<td>v ← +Infinity</td>
<td>v ← src1</td>
<td>v ← +Infinity</td>
<td>v ← src1</td>
<td>v ← dQNaN</td>
<td>v ← +Infinity</td>
<td>v ← src2</td>
<td>v ← +Infinity</td>
</tr>
<tr>
<td>src1</td>
<td>src2</td>
<td>v ← +Infinity</td>
<td>v ← src1</td>
<td>v ← +Infinity</td>
<td>v ← src1</td>
<td>v ← dQNaN</td>
<td>v ← +Infinity</td>
<td>v ← src2</td>
<td>v ← +Infinity</td>
</tr>
<tr>
<td>src1</td>
<td>src2</td>
<td>v ← +Infinity</td>
<td>v ← src1</td>
<td>v ← +Infinity</td>
<td>v ← src1</td>
<td>v ← dQNaN</td>
<td>v ← +Infinity</td>
<td>v ← src2</td>
<td>v ← +Infinity</td>
</tr>
<tr>
<td>src1</td>
<td>src2</td>
<td>v ← +Infinity</td>
<td>v ← src1</td>
<td>v ← +Infinity</td>
<td>v ← src1</td>
<td>v ← dQNaN</td>
<td>v ← +Infinity</td>
<td>v ← src2</td>
<td>v ← +Infinity</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

xsaddqp VRT,VRA,VRB (RO=0)
xsaddqpo VRT,VRA,VRB (RO=1)

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.

Otherwise, 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].

1. The quad-precision default Quiet NaN is the value, 0x7FFFFFFF_8000_0000_0000_0000_0000_0000_0000.
Chapter 7. Vector-Scalar Floating-Point Operations

Table 53. 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.

### Explanation:
- **src1** The quad-precision floating-point value in VSR[VR[A+32]].
- **src2** 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 and opposite signs).
- **add(x,y)** The floating-point value y is added 1 to the floating-point value x. Return the normalized 1 sum, having unbounded significand precision and exponent range.
- **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.
Let $X_A$ be the sum $32 	imes A + A$.
Let $X_B$ be the sum $32 	imes B + B$.

Let $src_1$ be the double-precision floating-point value in doubleword element 0 of $VSR[X_A]$.

Let $src_2$ be the double-precision floating-point value in doubleword element 0 of $VSR[X_B]$.

The exponent of $src_1$ is compared with the exponent of $src_2$. The result of the compare is placed into FPCC and CR field BF.

**Special Registers Altered:**
- CR field BF
- FPCC

*Programming Note*

This instruction can be used to operate on single-precision source operands.

---

**VSR Data Layout for xscmpexpdp**

<table>
<thead>
<tr>
<th>src1</th>
<th>VSR[X_A].dword[0]</th>
<th>unused</th>
</tr>
</thead>
<tbody>
<tr>
<td>src2</td>
<td>VSR[X_B].dword[0]</td>
<td>unused</td>
</tr>
</tbody>
</table>
Chapter 7. Vector-Scalar Floating-Point Operations

### VSX Scalar Compare Exponents Quad-Precision X-form

**xscmpexpqpp**  BF,VRA,VRB

<table>
<thead>
<tr>
<th>63</th>
<th>BF</th>
<th>/</th>
<th>VRA</th>
<th>VRB</th>
<th>164</th>
<th>/</th>
</tr>
</thead>
</table>

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 xscmpexpqpp

| VSR[VRA+32] | src1 |
| VSR[VRB+32] | src2 |

```plaintext
if MSR.VSX=0 then VSX_Unavailable()

reset_flags()

src1 ← VSR[VRA+32]
src2 ← VSR[VRB+32]

src1.exponent ← EXTZ(src1.bit[1:35])
src2.exponent ← EXTZ(src2.bit[1:35])
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)
ue_flag ← src1.class.NaN | src2.class.NaN

CR.bit[4×BF+32] ← FPSCR.FL ← !ue_flag & lt_flag
CR.bit[4×BF+33] ← FPSCR.FG ← !ue_flag & gt_flag
CR.bit[4×BF+34] ← FPSCR.FE ← src1.class.NaN & eq_flag
CR.bit[4×BF+35] ← FPSCR.FU ← ue_flag
```
VSX Scalar Compare Equal Double-Precision XX3-form

xscmppeqdp XT,XA,XB

<table>
<thead>
<tr>
<th>T</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</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[32×AX+A].

Let src2 be the double-precision floating-point value in doubleword 0 of VSR[32×BX+B].

If src1 or src2 is a SNaN, an Invalid Operation exception occurs.

src1 is compared to src2.

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
VSX Scalar Compare Greater Than or Equal Double-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 $32 \times BX + B$.

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

### Definitions
- $XT$: The value $32 \times TX + T$.
- $XA$: The value $32 \times AX + A$.
- $XB$: The value $32 \times BX + B$.
- $src1$: The double-precision floating-point value in doubleword 0 of VSR[$XA$].
- $src2$: The double-precision floating-point value in doubleword 0 of VSR[$XB$].
- VXSNAN: A flag for signaling a NaN.
- VXVC: A flag for signaling a trap-enabled Invalid Operation.

### Code Snippet
```assembly
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") | (src2.class="SNaN") then do
  vxsnan_flag ← 0b1
  if(FPSCR.VE=0) then vxvc_flag ← 0b1
end
else
  vxvc_flag ← FPSCR.VE & (vxsnan_flag | vxvc_flag)
if (vxsnan_flag=1) SetFX(FPSCR.VXSNAN)
if (vxvc_flag=1) SetFX(FPSCR.VXVC)
if (vxvc_flag=0) then do
  if bfp_COMPARE_GE(src1, src2)=1 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
```
VSX Scalar Compare Greater Than Double-Precision XX3-form

xscmpgtdp XT,XA,XB

Let XT be the value 32xTX + T.
Let XA be the value 32xAX + A.
Let XB be the value 32xBX + 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
VSX Scalar Compare Ordered
Double-Precision XX3-form

xscmpodp BF,XA,XB

<table>
<thead>
<tr>
<th>60</th>
<th>BF</th>
<th>/</th>
<th>A</th>
<th>B</th>
<th>43</th>
<th>CR//</th>
<th>F</th>
<th>FX</th>
<th>VXSNAN</th>
<th>VXVC</th>
</tr>
</thead>
</table>
| XA | ← AX || A
| XB | ← BX || B
| reset_xflags() |
| src1 | ← VSR[XA](0:63)
| src2 | ← VSR[XB](0:63)

if( IsSNaN(src1) | IsSNaN(src2) ) then do
  vxsnan_flag ← 0b1
  if(VE=0) then vxvc_flag ← 0b1
end
else if( IsQNaN(src1) | IsQNaN(src2) ) then vxvc_flag = 0b1

FL ← CompareLTDP(src1,src2)
FG ← CompareGTDP(src1,src2)
FE ← CompareEQDP(src1,src2)
FU ← IsNAN(src1) | IsNAN(src2)
CR[BF] ← FL || FG || FE || FU
if(vxsnan_flag) then SetFX(VXSNAN)
if(vxvc_flag) then SetFX(VXVC)

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.

See Table 54, “Actions for xscmpodp - Part 1: Compare Ordered,” on page 528.

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, 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.

See Table 55, “Actions for xscmpodp - Part 2: Result,” on page 528.
### Table 54. Actions for xscmpodp - Part 1: Compare Ordered

<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>ccc–0b0010</td>
<td>ccc–0b1000</td>
<td>ccc–0b1000</td>
<td>ccc–0b1000</td>
<td>ccc–0b1000</td>
<td>cc–0b0001</td>
<td>vxvc_flag=1</td>
<td>vxvc_flag=1</td>
</tr>
<tr>
<td>~NZF</td>
<td>ccc–0b0100</td>
<td>ccc–C(src1,src2)</td>
<td>ccc–0b1000</td>
<td>ccc–0b1000</td>
<td>ccc–0b1000</td>
<td>cc–0b0001</td>
<td>vxvc_flag=1</td>
<td>vxvc_flag=1</td>
</tr>
<tr>
<td>~Zero</td>
<td>ccc–0b0100</td>
<td>ccc–0b0100</td>
<td>ccc–0b0010</td>
<td>ccc–0b0010</td>
<td>ccc–0b0010</td>
<td>cc–0b0001</td>
<td>vxvc_flag=1</td>
<td>vxvc_flag=1</td>
</tr>
<tr>
<td>+Zero</td>
<td>ccc–0b0100</td>
<td>ccc–0b0100</td>
<td>ccc–0b0010</td>
<td>ccc–0b0010</td>
<td>ccc–0b0010</td>
<td>cc–0b0001</td>
<td>vxvc_flag=1</td>
<td>vxvc_flag=1</td>
</tr>
<tr>
<td>+Infinity</td>
<td>ccc–0b0100</td>
<td>ccc–0b0100</td>
<td>ccc–0b0010</td>
<td>ccc–0b0010</td>
<td>ccc–0b0010</td>
<td>cc–0b0001</td>
<td>vxvc_flag=1</td>
<td>vxvc_flag=1</td>
</tr>
<tr>
<td>QNaN</td>
<td>cc–0b0001</td>
<td>cc–vxsnan_flag=1</td>
<td>cc–vxsnan_flag=1</td>
<td>cc–vxsnan_flag=1</td>
<td>cc–vxsnan_flag=1</td>
<td>cc–0b0001</td>
<td>vxvc_flag=1</td>
<td>vxvc_flag=1</td>
</tr>
<tr>
<td>SNaN</td>
<td>cc–0b0001</td>
<td>cc–vxsnan_flag=1</td>
<td>cc–vxsnan_flag=1</td>
<td>cc–vxsnan_flag=1</td>
<td>cc–vxsnan_flag=1</td>
<td>cc–0b0001</td>
<td>vxvc_flag=1</td>
<td>vxvc_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].
- **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 55. 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 0 0</td>
<td>FPCC=cc, CR[BF]=cc</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 1</td>
<td>FPCC=cc, CR[BF]=cc, fx(VXVC)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1 0</td>
<td>FPCC=cc, CR[BF]=cc, fx(VXSNAN), fx(VXVC)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1 1</td>
<td>FPCC=cc, CR[BF]=cc, fx(VXSNAN), error()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 1</td>
<td>FPCC=cc, CR[BF]=cc, fx(VXVC), error()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 1 1</td>
<td>FPCC=cc, CR[BF]=cc, fx(VXSNAN), error()</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Explanation:**
- **VE** The values depend on the condition.
- **cc** The 4-bit result as defined in Table 54.
- **fx(x)** FX is set to 1 if x=0. x is set to 1.
- **error()** The system error handler is invoked for the trap-enabled exception if the FE0 and FE1 bits in the Machine State Register are set to any mode other than the ignore-exception mode.
- **FX** Floating-Point Summary Exception status flag, FPSCR_FX
- **VXSNAN** Floating-Point Invalid Operation Exception (SNaN) status flag, FPSCR_VXSNAN. See Section 7.4.1.
- **VXC** Floating-Point Invalid Operation Exception (Invalid Compare) status flag, FPSCR_VXC. See Section 7.4.1.
### VSX Scalar Compare Ordered Quad-Precision X-form

**xscmpoqp** BF,VRA,VRB

<table>
<thead>
<tr>
<th></th>
<th>63</th>
<th>6</th>
<th>9</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>132</th>
<th>31</th>
</tr>
</thead>
</table>

Let \( s{rc1} \) be the floating-point value in \( VSR[VRA+32] \) represented in quad-precision format.

Let \( s{rc2} \) be the floating-point value in \( VSR[VRB+32] \) represented in quad-precision format.

\( s{rc1} \) is compared to \( s{rc2} \).

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 \( s{rc1} \) is less than \( s{rc2} \).

Bit 1 of CR field BF and FG are set to indicate if \( s{rc1} \) is greater than \( s{rc2} \).

Bit 2 of CR field BF and FE are set to indicate if \( s{rc1} \) is equal to \( s{rc2} \).

Bit 3 of CR field BF and FU are set to indicate unordered (i.e., \( s{rc1} \) or \( s{rc2} \) is a NaN).

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
- FPCC FX VXSNAN VXVC

#### VSR Data Layout for xscmpoqp

- **VSR[VRA+32]**
  - \( s{rc1} \)
- **VSR[VRB+32]**
  - \( s{rc2} \)
VSX Scalar Compare Unordered
Double-Precision XX3-form

xscmpudp BF,XA,XB

<table>
<thead>
<tr>
<th></th>
<th>60</th>
<th>BF</th>
<th>A</th>
<th>B</th>
<th>35</th>
</tr>
</thead>
<tbody>
<tr>
<td>XA</td>
<td>AX</td>
<td></td>
<td>A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XB</td>
<td>BX</td>
<td></td>
<td>B</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

reset_eflags()

src1 ← VSR[XA][0:63]
src2 ← VSR[XB][0:63]

if( IsSNaN(src1) | IsSNaN(src2) ) then vxsnan_flag ← 1

FL ← CompareLTDP(src1,src2)
FG ← CompareGTDP(src1,src2)
FE ← CompareEQDP(src1,src2)
FU ← IsNAN(src1) | IsNAN(src2)
CR[BF] ← FL || FG || FE || FU
if(vxsnan_flag) then SetFX(VXSNAN)

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.

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.

See Table 57, “Actions for xscmpudp - Part 2: Result,” on page 531.

Special Registers Altered
CR[BF]
FPCC FX VXSNAN

Programming Note
This instruction can be used to operate on single-precision source operands.
## Table 56. Actions for xscmpudp - Part 1: Compare Unordered

<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>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>vxsnan_flag = 1</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 = 0b1000</td>
<td>cc = 0b0001</td>
<td>vxsnan_flag = 1</td>
</tr>
<tr>
<td>–Zero</td>
<td>cc = 0b0100</td>
<td>cc = 0b0010</td>
<td>cc = 0b0010</td>
<td>cc = 0b0010</td>
<td>cc = 0b0010</td>
<td>cc = 0b0010</td>
<td>cc = 0b0010</td>
<td>vxsnan_flag = 1</td>
</tr>
<tr>
<td>+Zero</td>
<td>cc = 0b0100</td>
<td>cc = 0b0010</td>
<td>cc = 0b0010</td>
<td>cc = 0b0010</td>
<td>cc = 0b0010</td>
<td>cc = 0b0010</td>
<td>cc = 0b0010</td>
<td>vxsnan_flag = 1</td>
</tr>
<tr>
<td>+NZF</td>
<td>cc = 0b0100</td>
<td>cc = 0b0100</td>
<td>cc = 0b0100</td>
<td>cc = 0b0100</td>
<td>cc = 0b0100</td>
<td>cc = 0b0100</td>
<td>cc = 0b0100</td>
<td>vxsnan_flag = 1</td>
</tr>
<tr>
<td>+Infinity</td>
<td>cc = 0b0100</td>
<td>cc = 0b0100</td>
<td>cc = 0b0100</td>
<td>cc = 0b0100</td>
<td>cc = 0b0100</td>
<td>cc = 0b0100</td>
<td>cc = 0b0100</td>
<td>vxsnan_flag = 1</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>vxsnan_flag = 1</td>
</tr>
<tr>
<td>SNaN</td>
<td>cc = 0b0001</td>
<td>vxsnan_flag = 1</td>
<td>cc = 0b0001</td>
<td>vxsnan_flag = 1</td>
<td>cc = 0b0001</td>
<td>vxsnan_flag = 1</td>
<td>cc = 0b0001</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].
- **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 57. Actions for xscmpudp - Part 2: Result

<table>
<thead>
<tr>
<th>VE</th>
<th>Returned Results and Status Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>FPCC ← cc, CR[BF] ← cc</td>
</tr>
<tr>
<td>1</td>
<td>FPCC ← cc, CR[BF] ← cc, fx(VXSNAN)</td>
</tr>
<tr>
<td>1</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 56.
- **fx(x)**: FX is set to 1 if x=0. x is set to 1.
- **error()**: The system error handler is invoked for the trap-enabled exception if the FE0 and FE1 bits in the Machine State Register are set to any mode other than the ignore-exception mode.
- **FX**: Floating-Point Summary Exception status flag, FPSCR FX.
- **VXSNAN**: Floating-Point Invalid Operation Exception (SNaN) status flag, FPSCR VXSNAN. See Section 7.4.1.
**VSX Scalar Compare Unordered Quad-Precision X-form**

Let \( \text{src1} \) be the floating-point value in \( \text{VSR[VRA+32]} \) represented in quad-precision format.

Let \( \text{src2} \) be the floating-point value in \( \text{VSR[VRB+32]} \) represented in quad-precision format.

\( \text{src1} \) is compared to \( \text{src2} \).

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 \( \text{src1} \) is less than \( \text{src2} \).

Bit 1 of CR field BF and FG are set to indicate if \( \text{src1} \) is greater than \( \text{src2} \).

Bit 2 of CR field BF and FE are set to indicate if \( \text{src1} \) is equal to \( \text{src2} \).

Bit 3 of CR field BF and FU are set to indicate unordered (i.e., \( \text{src1} \) or \( \text{src2} \) is a NaN).

If either of the operands is a Signaling NaN, an Invalid Operation exception occurs and \( \text{VXSNAN} \) is set to 1.

**Special Registers Altered:**

- CR field BF
- FPCC
- FX
- VXSNAN

**VSR Data Layout for xscmpuqp**

<table>
<thead>
<tr>
<th>VSR[VRA+32]</th>
<th>src1</th>
</tr>
</thead>
<tbody>
<tr>
<td>VSR[VRB+32]</td>
<td>src2</td>
</tr>
</tbody>
</table>

### VSX Vector Data Layout for xscmpuqp

<table>
<thead>
<tr>
<th>63</th>
<th>BF</th>
<th>VRA</th>
<th>VRB</th>
<th>644</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>9</td>
<td>11</td>
<td>18</td>
</tr>
</tbody>
</table>

If MSR.VSX=0 then VSX_Unavailable()

reset_xflags()

\( \text{src1} \) \( \leftarrow \) bfp_CONVERT_FROM_BFP128(VSR[VRA+32])

\( \text{src2} \) \( \leftarrow \) bfp_CONVERT_FROM_BFP128(VSR[VRB+32])

vxsnan_flag \( \leftarrow \) src1.class.SNaN | src2.class.SNaN

cc.bit[0] \( \leftarrow \) bfp_COMPARE_LT(src1,src2)

cc.bit[1] \( \leftarrow \) bfp_COMPARE_GT(src1,src2)

cc.bit[2] \( \leftarrow \) bfp_COMPARE_EQ(src1,src2)

cc.bit[3] \( \leftarrow \) src1.class.SNaN | src1.class.QNaN | src2.class.SNaN | src2.class.QNaN

if(vxsnan_flag) then SetFX(FPSCR.VXSNAN)

FPSCR.FPCC \( \leftarrow \) cc

CR.field[BF] \( \leftarrow \) cc
VSX Scalar Copy Sign Double-Precision XX3-form

\[
xscpsgndp \quad XT,XA,XB
\]

\[
\begin{array}{cccccc}
60 & 6 & 11 & 16 & 21 & 176 \\
\hline
XT & \Leftarrow TX || T & A & B & \text{undefined} & \text{undefined} \\
XA & \Leftarrow AX || A & B & \text{undefined} & \text{undefined} & \text{undefined} \\
XB & \Leftarrow BX || B & \text{undefined} & \text{undefined} & \text{undefined} & \text{undefined} \\
\end{array}
\]

Let \( XT \) be the value \( 32 \times TX + T \).
Let \( XA \) be the value \( 32 \times AX + A \).
Let \( XB \) be the value \( 32 \times BX + B \).

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 undefined.

\textbf{Special Registers Altered}: None

\textbf{VSR Data Layout for xscpsgndp}

\begin{enumerate}
\item src1 = VSR\( [XA] \)
\item src2 = VSR\( [XB] \)
\item tgt = VSR\( [XT] \)
\end{enumerate}

\textbf{Programming Note}

This instruction can be used to operate on single-precision source operands.

VSX Scalar Copy Sign Quad-Precision X-form

\[
xscpsgnqp \quad VRT,VRA,VRB
\]

\[
\begin{array}{cccccc}
63 & 6 & 11 & 16 & 21 & 100 \\
\hline
\text{if MSR.VSX=0 then VSX_Unavailable} & & & & & \\
src1 & \Leftarrow VSR[VRA+32] & | 0x8000_0000_0000_0000_0000_0000_0000_0000 & \text{undefined} & \\
src2 & \Leftarrow VSR[VRB+32] & | 0x7FFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF & \text{undefined} & \\
VSR[VRT+32] & \Leftarrow src1 | src2 & & & & \\
\end{array}
\]

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 \( src1 \).

\textbf{Special Registers Altered}: None

\textbf{VSR Data Layout for xscpsgnqp}

\begin{enumerate}
\item src1
\item src2
\item tgt
\end{enumerate}
VSX Scalar Convert with round
Double-Precision to Half-Precision format
XX2-form

src

<table>
<thead>
<tr>
<th>0</th>
<th>6</th>
<th>T</th>
<th>17</th>
<th>B</th>
<th>347</th>
<th>60</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>

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 undefined.

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
FX VXSNAN OX UX XX

Programming Note

This instruction can be used to operate on a single-precision source operand.
VSX Scalar Convert Double-Precision to Quad-Precision format X-form

Let \( \text{src} \) be the floating-point value in doubleword element 0 of \( \text{VSR}[\text{VRB}+32] \) represented in double-precision format.

\( \text{src} \) is placed into \( \text{VSR}[\text{VRT}+32] \) 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, \( \text{VSR}[\text{XT}] \) and \( \text{FPRF} \) are not modified.

Special Registers Altered:

\( \text{FPRF} \), \( \text{FR} \) (set to 0), \( \text{FI} \) (set to 0)
\( \text{FX} \), \( \text{VXSNAN} \)

VSR Data Layout for \text{xscvdpqp}

<table>
<thead>
<tr>
<th>63</th>
<th>VRT</th>
<th>22</th>
<th>VRB</th>
<th>836</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>11</td>
<td>10</td>
<td>11</td>
<td>836</td>
</tr>
</tbody>
</table>

if MSR.VSX=0 then VSX_Unavailable()  
src ← bfp_CONVERT_FROM_BFP64(\( \text{VSR}[\text{VRB}+32].\text{dword}[0] \))  
if src.class.SNaN then  
result ← bfp_CONVERT_TO_BFP128(bfp_QUIET(src))  
else  
result ← bfp_CONVERT_TO_BFP128(src)  

vxsnan_flag ← src.class.SNaN  
if(vxsnan_flag) then SetFX(FPSCR.VXSNAN)  
vex_flag ← FPSCR.VE & vxsnan_flag  
if vex_flag then  
\( \text{VSR}[\text{VRT}+32] \) ← result  
FPSCR.FPRF ← fprf_CLASS_BFP128(result)  
end  
FPSCR.FR ← 0  
FPSCR.FI ← 0

VSR Data Layout for \text{xscvdpqp}

<table>
<thead>
<tr>
<th>src.dword[0]</th>
<th>unused</th>
</tr>
</thead>
<tbody>
<tr>
<td>tgt</td>
<td></td>
</tr>
</tbody>
</table>

Chapter 7. Vector-Scalar Floating-Point Operations
VSX Scalar Convert with round
Double-Precision to Single-Precision format
XX2-form

xscvdpsp XT,XB

<table>
<thead>
<tr>
<th></th>
<th>T</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>265</th>
<th>BX</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>60</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>11</td>
<td></td>
</tr>
</tbody>
</table>

reset_xflags()
src ← VSR[32×BX+B].dword[0]
result ← ConvertDPtoSP(src)
if(vxsnan_flag) then SetFX(FPSCR.VXSNAN)
if(xx_flag) then SetFX(FPSCR.XX)
if(xx_flag) then SetFX(FPSCR.OX)
if(xx_flag) then SetFX(FPSCR.UX)
vec_flag ← FPSCR.VE & vxsnan_flag
if( ~vec_flag ) then do
VSR[32×TX+T].word[0] ← result
VSR[32×TX+T].word[1] ← 0xUUUU_UUUU
VSR[32×TX+T].word[2] ← 0xUUUU_UUUU
VSR[32×TX+T].word[3] ← 0xUUUU_UUUU
FPSCR.FR ← ClassSP(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 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 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.

See Table 50, “Scalar Floating-Point Intermediate Result Handling,” on page 515.

The result is placed into word element 0 of VSR[XT] in single-precision format.

The contents of word elements 1, 2, and 3 of VSR[XT] are undefined.

FPFR 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 FPFR are not modified, and FR and FI are set to 0.

See Table 51, “VSX Scalar Floating-Point Final Result,” on page 516.

Special Registers Altered
FPFR FR FI FX OX UX XX VXSNAN

VSR Data Layout for xscvdpsp
src = VSR[XB]
tgt = VSR[XT]

<table>
<thead>
<tr>
<th></th>
<th>3</th>
<th>64</th>
<th>127</th>
</tr>
</thead>
<tbody>
<tr>
<td>DP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>unused</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>32</th>
<th>64</th>
<th>127</th>
</tr>
</thead>
<tbody>
<tr>
<td>SP</td>
<td></td>
<td>undefined</td>
<td>undefined</td>
<td></td>
</tr>
</tbody>
</table>

Programming Note
This instruction can be used to operate on a single-precision source operand.
**VSX Scalar Convert Scalar Single-Precision to Vector Single-Precision format Non-signalling XX2-form**

\[ xscvdpspn \quad XT, XB \]

<table>
<thead>
<tr>
<th>60</th>
<th>T</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>30-31</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **reset_xflags()**: reset_xflags()
- **src**: \( VSR[32 \times BX + B].dword[0] \)
- **result**: \( ConvertDPtoSP_NS[src] \)
- **XT, XB**: \( VSR[32 \times TX + T].word[0] \)
- **result**: \( ConvertDPtoSP_NS[src] \)
- **XT, XB**: \( VSR[32 \times TX + T].word[1] \)
- **result**: \( ConvertDPtoSP_NS[src] \)
- **XT, XB**: \( VSR[32 \times TX + T].word[2] \)
- **result**: \( ConvertDPtoSP_NS[src] \)
- **XT, XB**: \( VSR[32 \times TX + T].word[3] \)
- **result**: \( ConvertDPtoSP_NS[src] \)

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 element \( 0 \) of \( VSR[XT] \) in single-precision format.

The contents of word elements \( 1, 2, \) and \( 3 \) of \( VSR[XT] \) are undefined.

**Special Registers Altered**
None

**VSR Data Layout for xscvdpspn**

- **src**: \( VSR[XB] \)
- **tgt**: \( VSR[XT] \)

**Programming Note**

- **xscvdpspn** 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.

---

**VSX Scalar Convert with round to zero Double-Precision to Signed Doubleword format XX2-form**

\[ xscvdpsxds \quad XT, XB \]

<table>
<thead>
<tr>
<th>60</th>
<th>T</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>344</th>
<th>333</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **XT**: \( TX || T \)
- **XB**: \( BX || B \)
- **reset_xflags()**: reset_xflags()
- **src**: \( VSR[XB]{0:63} \)
- **result**: \( ConvertDPtoSD(src) \)
- **fr**: \( 0bUUUUU \)
- **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 \( VSR[XB] \).

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 \( 2^{63} \), the result is \( 0x7FFF_FFFF_FFFF_FFFF \) and \( VXCVI \) is set to 1.

Otherwise, \( src \) is rounded to a floating-point integer using the rounding mode Round Toward Zero.

If a trap-enabled invalid operation exception occurs,
- \( VXSNAN \) 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 undefined.
- 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 58.

Special Registers Altered

FPRF=0bUUUUU FR FI FX XX VXSNAN VXCVI

VSR Data Layout for xscvdpsxds

<table>
<thead>
<tr>
<th>src = VSR[XB]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>DP</td>
</tr>
<tr>
<td>unused</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>tgt = VSR[XT]</th>
</tr>
</thead>
<tbody>
<tr>
<td>SD</td>
</tr>
<tr>
<td>undefined</td>
</tr>
</tbody>
</table>

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.
### Returned Results and Status Setting

<table>
<thead>
<tr>
<th>src</th>
<th>XE</th>
<th>VE</th>
<th>Inexact? (RoundToDPintegerTrunc(src))</th>
<th>T(x)</th>
<th>FR</th>
<th>FI</th>
<th>FX</th>
<th>error()</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nmin - 1</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>T(Nmin)</td>
<td>FR = 0, FI = 0, fx(VXCVI)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>FR = 0, FI = 0, fx(VXCVI), error()</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nmin - 1 &lt; src &lt; Nmin</td>
<td></td>
<td></td>
<td>yes</td>
<td>T(Nmin)</td>
<td>FR = 0, FI = 1, fx(XX)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>T(Nmin)</td>
<td>FR = 0, FI = 1, fx(XX), error()</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>src = Nmin</td>
<td></td>
<td></td>
<td>-</td>
<td>-</td>
<td>no</td>
<td>T(Nmin)</td>
<td>FR = 0, FI = 0</td>
<td></td>
</tr>
<tr>
<td>Nmin &lt; src &lt; Nmax</td>
<td></td>
<td></td>
<td>-</td>
<td>-</td>
<td>no</td>
<td>T(ConvertDPtoSD(RoundToDPintegerTrunc(src)))</td>
<td>FR = 0, FI = 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>yes</td>
<td>T(ConvertDPtoSD(RoundToDPintegerTrunc(src)))</td>
<td>FR = 0, FI = 1, fx(XX)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>yes</td>
<td>T(ConvertDPtoSD(RoundToDPintegerTrunc(src)))</td>
<td>FR = 0, FI = 1, fx(XX), error()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>src = Nmax</td>
<td></td>
<td></td>
<td>-</td>
<td>-</td>
<td>no</td>
<td>T(Nmax)</td>
<td>FR = 0, FI = 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>-</td>
<td>-</td>
<td>no</td>
<td>(Note: This case cannot occur as Nmax is not representable in DP format but is included here for completeness.)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nmax &lt; src &lt; Nmax + 1</td>
<td></td>
<td></td>
<td>-</td>
<td>0</td>
<td>yes</td>
<td>T(Nmax)</td>
<td>FR = 0, FI = 1, fx(XX)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>yes</td>
<td>T(Nmax)</td>
<td>FR = 0, FI = 1, fx(XX), error()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>src = Nmax + 1</td>
<td></td>
<td></td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>T(Nmax)</td>
<td>FR = 0, FI = 0, fx(VXCVI)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>FR = 0, FI = 0, fx(VXCVI), error()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>src is a QNaN</td>
<td></td>
<td></td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>T(Nmax)</td>
<td>FR = 0, FI = 0, fx(VXCVI)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>FR = 0, FI = 0, fx(VXCVI), error()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>src is a SNaN</td>
<td></td>
<td></td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>T(Nmax)</td>
<td>FR = 0, FI = 0, fx(VXCVI), fx(VXSNAN)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>FR = 0, FI = 0, fx(VXCVI), fx(VXSNAN), error()</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Table 58. Actions for xscvdpsxds

**Explanation:**

- \( fx(x) \): FX is set to 1 if \( x = 0 \). \( x \) is set to 1.
- \( error() \): The system error handler is invoked for the trap-enabled exception if the FE0 and FE1 bits in the Machine State Register are set to any mode other than the ignore-exception mode.
- \( N_{min} \): The smallest signed integer doubleword value, \(-2^{31} \times 0x8000_0000_0000_0000\).
- \( N_{max} \): The largest signed integer doubleword value, \(2^{31} - 1 \times 0x7FFF_FFFF_FFFF_FFFF\).
- src: The double-precision floating-point value in doubleword element 0 of VSR[XB].
- 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 undefined.
VSX Scalar Convert with round to zero
Double-Precision to Signed Word format
XX2-form

xscvdpsxws XT,XB

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 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 $2^{31} - 1$, 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.

If a trap-enabled invalid operation exception occurs,

- VSR[XT] and FPRF are not modified
- FR and FI are set to 0.
Chapter 7. Vector-Scalar Floating-Point Operations

541

Table 59. Actions for xscvdpsxws

<table>
<thead>
<tr>
<th>VE</th>
<th>XE</th>
<th>Inexact? (RoundToDPintegerTrunc &amp; src)</th>
<th>Returned Results and Status Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>src ≤ Nmin</td>
<td>0</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>1</td>
<td>–</td>
<td>FR=0, Fl=0, fx(VXCVI), error()</td>
<td></td>
</tr>
<tr>
<td>Nmin &lt; src &lt; Nmin</td>
<td>0</td>
<td>yes</td>
<td>T(Nmin), FR=0, Fl=1, fx(XX)</td>
</tr>
<tr>
<td>1</td>
<td>yes</td>
<td>T(Nmin), FR=0, Fl=1, fx(XX), error()</td>
<td></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>–</td>
<td>no</td>
</tr>
<tr>
<td>0</td>
<td>yes</td>
<td>T(ConvertDPtoSW(RoundToDPintegerTrunc(src))), FR=0, Fl=1, fx(XX)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>yes</td>
<td>T(ConvertDPtoSW(RoundToDPintegerTrunc(src))), FR=0, Fl=1, fx(XX), error()</td>
<td></td>
</tr>
<tr>
<td>src = Nmax</td>
<td>–</td>
<td>–</td>
<td>no</td>
</tr>
<tr>
<td>Nmax &lt; src &lt; Nmax+1</td>
<td>–</td>
<td>–</td>
<td>no</td>
</tr>
<tr>
<td>0</td>
<td>yes</td>
<td>T(ConvertDPtoSW(RoundToDPintegerTrunc(src))), FR=0, Fl=1, fx(XX)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>yes</td>
<td>T(ConvertDPtoSW(RoundToDPintegerTrunc(src))), FR=0, Fl=1, fx(XX), error()</td>
<td></td>
</tr>
<tr>
<td>src = Nmax+1</td>
<td>0</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>1</td>
<td>–</td>
<td>FR=0, Fl=0, fx(VXCVI), error()</td>
<td></td>
</tr>
<tr>
<td>src is a QNaN</td>
<td>0</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>1</td>
<td>–</td>
<td>FR=0, Fl=0, fx(VXCVI), error()</td>
<td></td>
</tr>
<tr>
<td>src is a SNaN</td>
<td>0</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>1</td>
<td>–</td>
<td>FR=0, Fl=0, fx(VXCVI), fx(VXSNAN), error()</td>
<td></td>
</tr>
</tbody>
</table>

Explanations:

- **fx(x)**: FX is set to 1 if x=0. x is set to 1.
- **error()**: The system error handler is invoked for the trap-enabled exception if the FE0 and FE1 bits in the Machine State Register are set to any mode other than the ignore-exception mode.
- **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 double-precision floating-point value in doubleword element 0 of VSR[XB].
- **T(x)**: The signed integer word value x is placed in word element 1 of VSR[XT]. The contents of word elements 0, 2, and 3 of VSR[XT] are undefined.
VSX Scalar Convert with round to zero
Double-Precision to Unsigned Doubleword format XX2-form

\[
xscvdpuxd \quad XT, XB
\]

<table>
<thead>
<tr>
<th>60</th>
<th>T</th>
<th>B</th>
<th>32B</th>
<th>64X</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
XT & \leftarrow TX || T \\
XB & \leftarrow BX || B \\
inc\_flag & \leftarrow 0b0 \\
reset\_flags() & \leftarrow ConvertDPtoUD(VSR[XB][0:63]) \\
\text{if} \{vxsnan\_flag\} & \text{then SetFX(VXSNAN)} \\
\text{if} \{vxcvi\_flag\} & \text{then SetFX(VXCVI)} \\
\text{if} \{xx\_flag\} & \text{then SetFX(XX)} \\
vex\_flag & \leftarrow VE \& (vxsnan\_flag \lor vxcvi\_flag) \\
\text{if} \{~vex\_flag\} & \text{then do} \\
VSR[XT] & \leftarrow result || 0xUUUU_UUUU_UUUU_UUUU \\
FPRF & \leftarrow 0bUUUUU \\
FR & \leftarrow inc\_flag \\
FI & \leftarrow xx\_flag \\
\text{end} \\
\text{else do} \\
FR & \leftarrow 0b0 \\
FI & \leftarrow 0b0 \\
\text{end}
\end{align*}
\]

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 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 \( 0xFFF_FFFF_FFFF_FFFF \) 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,

- \( VSR[XT] \) 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 undefined.
- \( FPRF \) 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 60.

**Special Registers Altered**

<table>
<thead>
<tr>
<th>FPRF</th>
<th>FR</th>
<th>FI</th>
<th>XX</th>
<th>VXSNAN</th>
<th>VXCVI</th>
</tr>
</thead>
<tbody>
<tr>
<td>0bUUUUU</td>
<td>0b0</td>
<td>0b0</td>
<td>0b0</td>
<td>0b0</td>
<td>0b0</td>
</tr>
</tbody>
</table>

**VSR Data Layout for xscvdpuxd**

<table>
<thead>
<tr>
<th>src = VSR[XB]</th>
</tr>
</thead>
<tbody>
<tr>
<td>DP</td>
</tr>
<tr>
<td>FR</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>tgt = VSR[XT]</th>
</tr>
</thead>
<tbody>
<tr>
<td>UD</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

---

**Programming Note**

This instruction can be used to operate on a single-precision source operand.

---

**Programming Note**

\( xscvdpuxd \) 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 \( RN \).
## Returned Results and Status Setting

<table>
<thead>
<tr>
<th>src</th>
<th>VE</th>
<th>XE</th>
<th>Inexact? (RoundToDPintegerTrunc $g$ src)</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, Fx=0, fx(VXCVI)</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>FR=0, Fx=0, fx(VXCVI), error()</td>
</tr>
<tr>
<td>Nmin-1 &lt; src &lt; Nmin</td>
<td>-</td>
<td>0</td>
<td>yes</td>
<td>T(Nmin), FR=0, Fx=1, fx(XX)</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>1</td>
<td>yes</td>
<td>T(Nmin), FR=0, Fx=1, fx(XX), error()</td>
</tr>
<tr>
<td>src = Nmin</td>
<td>-</td>
<td>-</td>
<td>no</td>
<td>T(Nmin), FR=0, Fx=0</td>
</tr>
<tr>
<td>Nmin &lt; src &lt; Nmax</td>
<td>-</td>
<td>no</td>
<td>T(ConvertDPtoUD(RoundToDPintegerTrunc(src))), FR=0, Fx=0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>yes</td>
<td>T(ConvertDPtoUD(RoundToDPintegerTrunc(src))), FR=0, Fx=1, fx(XX)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>yes</td>
<td>T(ConvertDPtoUD(RoundToDPintegerTrunc(src))), FR=0, Fx=1, fx(XX), error()</td>
<td></td>
</tr>
<tr>
<td>src = Nmax</td>
<td>-</td>
<td>-</td>
<td>no</td>
<td>T(Nmax), FR=0, Fx=0</td>
</tr>
<tr>
<td>Note: This case cannot occur as Nmax is not representable in DP format but is included here for completeness.</td>
<td></td>
<td></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), FR=0, Fx=1, fx(XX)</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>1</td>
<td>yes</td>
<td>T(Nmax), FR=0, Fx=1, fx(XX), error()</td>
</tr>
<tr>
<td>src = Nmax+1</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>T(Nmax), FR=0, Fx=0, fx(VXCVI)</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>FR=0, Fx=0, fx(VXCVI), error()</td>
</tr>
<tr>
<td>src is a QNaN</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>T(Nmax), FR=0, Fx=0, fx(VXCVI), fx(VXSNAN)</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>FR=0, Fx=0, fx(VXCVI), fx(VXSNAN), error()</td>
</tr>
<tr>
<td>src is a SNaN</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>T(Nmax), FR=0, Fx=0, fx(VXCVI), fx(VXSNAN), fx(VXSNAN)</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>FR=0, Fx=0, fx(VXCVI), fx(VXSNAN), fx(VXSNAN), error()</td>
</tr>
</tbody>
</table>

### Explanation:

- **fx(x)**: FX is set to 1 if $x=0$. $x$ is set to 1.
- **error()**: The system error handler is invoked for the trap-enabled exception if the FE0 and FE1 bits in the Machine State Register are set to any mode other than the ignore-exception mode.
- **Nmin**: The smallest unsigned integer doubleword value, 0 (0x0000_0000_0000_0000).
- **Nmax**: The largest unsigned integer doubleword value, 2^{64}-1 (0xFFF_FFFF_FFFF_FFFF).
- **src**: The double-precision floating-point value in doubleword element 0 of VSR[XB].
- **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 undefined.

### Table 60: Actions for xscvdpxuds
VSX Scalar Convert with round to zero
Double-Precision to Unsigned Word format
XX2-form

xscvdpuwxs XT,XB

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 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
\( 0xFF_FFFF \) 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.

Programming Note
This instruction can be used to operate on a
single-precision source operand.

Programming Note
\( xscvdpuwxs \) 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.
Inexact? (RoundToDPintegerTrunc(src))

<table>
<thead>
<tr>
<th>src</th>
<th>XE</th>
<th>Returned Results and Status Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nmin-1</td>
<td>-</td>
<td>T(Nmin), FR=0, FI=0, fx(VXCVI)</td>
</tr>
<tr>
<td>Nmin-1 &lt; src &lt; Nmin</td>
<td>-</td>
<td>FR=0, FI=1, fx(XX), error()</td>
</tr>
<tr>
<td>src = Nmin</td>
<td>-</td>
<td>no T(Nmin), FR=0, FI=0</td>
</tr>
<tr>
<td>Nmin &lt; src &lt; Nmax</td>
<td>-</td>
<td>no T(ConvertDPtoUW(RoundToDPintegerTrunc(src))), FR=0, FI=0</td>
</tr>
<tr>
<td>Nmax &lt; src &lt; Nmax+1</td>
<td>-</td>
<td>yes T(ConvertDPtoUW(RoundToDPintegerTrunc(src))), FR=0, FI=1, fx(XX)</td>
</tr>
<tr>
<td>src = Nmax</td>
<td>-</td>
<td>no T(Nmax), FR=0, FI=0</td>
</tr>
<tr>
<td>src in Nmax+1</td>
<td>-</td>
<td>yes T(Nmax), FR=0, FI=1, fx(XX), error()</td>
</tr>
<tr>
<td>src is a QNaN</td>
<td>-</td>
<td>no T(Nmax), FR=0, FI=0, fx(VXCVI), error()</td>
</tr>
<tr>
<td>src is a SNaN</td>
<td>-</td>
<td>no T(Nmax), FR=0, FI=0, fx(VXCVI), fx(VXSNAN), error()</td>
</tr>
</tbody>
</table>

**Explanation:**
- **fx(x)**: FX is set to 1 if x=0. x is set to 1.
- **error()**: The system error handler is invoked for the trap-enabled exception if the FE0 and FE1 bits in the Machine State Register are set to any mode other than the ignore-exception mode.
- **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 in doubleword element 0 of VSR[XB].
- **T(x)**: The unsinged integer word value x is placed in word element 1 of VSR[XT].
- **The contents of word elements 0, 2, and 3 of VSR[XT]** are undefined.

**Table 61. Actions for xscvdpuxws**
VSX Scalar Convert Half-Precision to Double-Precision format XX2-form

xscvhpdp XT,XB

<table>
<thead>
<tr>
<th>60</th>
<th>T</th>
<th>16</th>
<th>B</th>
<th>347</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>16</td>
<td>21</td>
<td>30</td>
</tr>
</tbody>
</table>

Let \( \text{XT} \) be the value \( 32 \times TX + T \).

Let \( \text{XB} \) be the value \( 32 \times BX + B \).

Let \( \text{src} \) be the half-precision floating-point value in the rightmost halfword of doubleword element 0 of VSR[\( \text{XB} \)].

If \( \text{src} \) is an SNaN, the result is the double-precision representation of that SNaN converted to a QNaN.

Otherwise, if \( \text{src} \) is a QNaN, the result is the double-precision representation of that QNaN.

Otherwise, if \( \text{src} \) is an Infinity, the result is the double-precision representation of Infinity with the same sign as \( \text{src} \).

Otherwise, if \( \text{src} \) is a Zero, the result is the double-precision representation of Zero with the same sign as \( \text{src} \).

Otherwise, if \( \text{src} \) is a denormal value, the result is the normalized double-precision representation of \( \text{src} \).

Otherwise, the result is the double-precision representation of \( \text{src} \).

The result is placed into doubleword element 0 of VSR[\( \text{XT} \)].

The contents of doubleword element 1 of VSR[\( \text{XT} \)] are undefined.

\( \text{FPRF} \) is set to the class and sign of the result as represented in half-precision.

If a trap-enabled invalid operation exception occurs, VSR[\( \text{XT} \)] and FPRF are not modified.

\( \text{FR} \) is set to 0, \( \text{FI} \) is set to 0.

Special Registers Altered:
\( \text{FPRF} \) FR (set to 0) FI (set to 0)
FX VXSNAN

VSR Data Layout for xscvhpdp

<table>
<thead>
<tr>
<th>( \text{src} )</th>
<th>unused</th>
<th>VSR[( \text{XB} ).hword[3]]</th>
<th>unused</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{tgt} )</td>
<td>VSR[( \text{XT} ).dword[0]]</td>
<td>undefined</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>48</td>
<td>64</td>
<td>127</td>
</tr>
</tbody>
</table>
VSX Scalar Convert with round
Quad-Precision to Double-Precision format
[using round to Odd] X-form

xscvqpdp  VRT,VRB  (RO=0)
xscvqpdp0 VRT,VRB  (RO=1)

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td>62</td>
<td>61</td>
<td>60</td>
<td>59</td>
<td>58</td>
</tr>
<tr>
<td>VRT</td>
<td>20</td>
<td>VRB</td>
<td>836</td>
<td>RO</td>
<td>57</td>
</tr>
</tbody>
</table>

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 ← bfp_CONVERT_TO_BFP64(rnd)

if(vxsnan_flag) then SetFX(FPSCR.VXSNAN)
if(ox_flag)     then SetFX(FPSCR.OX)
if(ux_flag)     then SetFX(FPSCR.UX)
if(xx_flag)     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

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 double-precision.

Otherwise, if src is an Infinity or a Zero, the result is src.

Otherwise, do the following.

If src is Tiny (i.e., the unbiased exponent is less than -1022) and UE=0, the significand is shifted right N bits, where N is the difference between -1022 and the unbiased exponent of src. The exponent of src is set to the value -1022.

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 double-precision (i.e., 11-bit exponent range and 53-bit significand precision) using the specified rounding mode.

See Table 50, “Scalar Floating-Point Intermediate Result Handling,” on page 515.

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 51, “VSX Scalar Floating-Point Final Result,” on page 516.

Special Registers Altered:

<table>
<thead>
<tr>
<th>FPRF</th>
<th>FR</th>
<th>FI</th>
<th>FX</th>
<th>VXSNAN</th>
<th>OX</th>
<th>UX</th>
<th>XX</th>
</tr>
</thead>
</table>

VSR Data Layout for xscvqpdp[o]

<table>
<thead>
<tr>
<th>VSR[VRT+32]</th>
<th>src</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>VSR[VRT+32]</th>
<th>tgt.dword[0]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0x0000_0000_0000_0000</td>
</tr>
</tbody>
</table>
VSX Scalar Convert with round to zero
Quad-Precision to Signed Doubleword format
X-form

```
xscvqpsdz  VRT,VRB
```

- **if MSR.VSX=0 then VSX_Unavailable()**
- **reset_fflags()**
- **src ← bfp_CONVERT_FROM_BFP128(VSR[VRB+32])**
- **if src.class.QNaN | src.class.SNaN then do**
  - **result ← 0x8000_0000_0000_0000**
  - **vxsnan_flag ← src.class.SNaN**
  - **vxcvi_flag ← 1**
- **else if src.class.Infinity then do**
  - **vxcvi_flag ← 1**
  - **if src.sign = 0 then**
    - **result ← 0x7FFF_FFFF_FFFF_FFFF**
  - **else**
    - **result ← 0x8000_0000_0000_0000**
- **else if src.class.Zero then**
  - **result ← 0x0000_0000_0000_0000**
- **else do**
  - **rnd ← bfp_ROUND_TO_INTEGER(0b001,src)**
  - **if bfp_COMPARE_GT(rnd, +2^63-1) then do**
    - **result ← 0x7FFF_FFFF_FFFF_FFFF**
    - **vxcvi_flag ← 1**
  - **else if bfp_COMPARE_LT(rnd, -2^63) then do**
    - **result ← 0x8000_0000_0000_0000**
    - **vxcvi_flag ← 1**
  - **else do**
    - **result ← bfp_CONVERT_TO_SI64(rnd)**
    - **if(xx_flag) then SetFX(FPSCR.XX)**
  - **end**
- **if(vxsnan_flag) then SetFX(FPSCR.VXSNAN)**
- **if(vxcvi_flag) then SetFX(FPSCR.VXCVI)**
- **vx_flag ← vxsnan_flag | vxcvi_flag**
- **ex_flag ← FPSCR.VE & vx_flag**
- **if ex_flag=0 then do**
  - **VSR[VRT+32].dword[0] ← result**
  - **VSR[VRT+32].dword[1] ← 0x0000_0000_0000_0000**
- **FPSCR.FR ← (vx_flag=0) & inc_flag**
  - **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 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.

FPFRF 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 FPFRF are not modified.

See Table 58, "Actions for xsvcdpsxsd," on page 539.

**Special Registers Altered:**

FPFRF (undefined) FR FI FX VXSNAN VXCVI XX

**VSR Data Layout for xscvqpsdz**

```
VSR[VRT+32]
```

<table>
<thead>
<tr>
<th>src</th>
<th>tgt.dword[0]</th>
<th>0x0000_0000_0000_0000</th>
</tr>
</thead>
</table>

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.

FPFRF 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 FPFRF are not modified.

See Table 58, "Actions for xsvcdpsxsd," on page 539.

**Special Registers Altered:**

FPFRF (undefined) FR FI FX VXSNAN VXCVI XX

**VSR Data Layout for xscvqpsdz**

```
VSR[VRT+32]
```

<table>
<thead>
<tr>
<th>src</th>
<th>tgt.dword[0]</th>
<th>0x0000_0000_0000_0000</th>
</tr>
</thead>
</table>

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.

FPFRF 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 FPFRF are not modified.

See Table 58, "Actions for xsvcdpsxsd," on page 539.

**Special Registers Altered:**

FPFRF (undefined) FR FI FX VXSNAN VXCVI XX

**VSR Data Layout for xscvqpsdz**

```
VSR[VRT+32]
```

<table>
<thead>
<tr>
<th>src</th>
<th>tgt.dword[0]</th>
<th>0x0000_0000_0000_0000</th>
</tr>
</thead>
</table>
### Returned Results and Status Setting

<table>
<thead>
<tr>
<th>src ≤ Nmin-1</th>
<th>0</th>
<th>–</th>
<th>T(Nmin), fr(0), fi(0), fprf(0bUUUUU), fx(VXCVI)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>–</td>
<td>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</td>
</tr>
<tr>
<td></td>
<td>–</td>
<td>1</td>
<td>yes</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>0</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td>–</td>
<td>1</td>
<td>yes</td>
</tr>
<tr>
<td>src = Nmax</td>
<td>–</td>
<td>–</td>
<td>no</td>
</tr>
<tr>
<td>Nmax &lt; src &lt; Nmax+1</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+1</td>
<td>0</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>src is a QNaN</td>
<td>0</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>

### Explanation:

\( T(x) \) places the value \( x \) into the target VSR.

\[
\begin{align*}
VSR[VAT+32].dword[0] & := x \\
VSR[VAT+32].dword[1] & := 0x0000_0000_0000_0000_0000_0000_0000_0000
\end{align*}
\]

- \( Nmin \): 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 quad-precision floating-point value in VSR[VRB+32].
- \( fx(x) \): \( FPSCR.FX \) is set to 1 if \( FPSCR.x=0 \). \( FPSCR.x \) 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 \( x \).
- \( error() \): 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 62. Actions for xcvqpsdz
VSX Scalar Convert with round to zero
Quad-Precision to Signed Word format X-form

xscvqpswz VRT,VRB

<table>
<thead>
<tr>
<th>63</th>
<th>VRT</th>
<th>9</th>
<th>VRB</th>
<th>836</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>0</td>
<td>9</td>
<td>16</td>
<td>836</td>
</tr>
</tbody>
</table>

- If MSR.VSX=0 then VSX_Unavailable();
- resetflags();
- src ← bfp_CONVERT_FROM_BFP128(VSR[VRB+32])

if src.class.QNaN | src.class.SNaN then do
  result ← 0xFFFF_FFFF_8000_0000
  vxsnan_flag ← src.class.SNaN
  vxcvi_flag ← 1
end
else if src.class.Infinity then do
  vxcvi_flag ← 1
  if src.sign = 0 then
    result ← 0x0000_0000_7FFF_FFFF
  else
    result ← 0xFFFF_FFFF_8000_0000
  end
else if src.class.Zero 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
    vxcvi_flag ← 1
  end
  else if bfp_COMPARE_LT(rnd, -231) then do
    result ← 0xFFFF_FFFF_8000_0000
    vxcvi_flag ← 1
  end
  else do
    result ← bfp_CONVERT_TO_SI64(rnd)
    if(xx_flag) then SetFX(FPSCR.XX)
  end
end

if vxsnan_flag then SetFX(FPSCR.VXSNAN)
if vxcvi_flag then SetFX(FPSCR.VXCVI)

vx_flag ← vxsnan_flag | vxcvi_flag
ex_flag ← FPSCR.VE & vx_flag

if ex_flag=0 then do
  VSR[VRT+32].dword[0] ← result
  VSR[VRT+32].dword[1] ← 0x0000_0000_0000_0000
  FPSCR.FPRF ← 0b00UUU
  FPSCR.FR ← 0
  FPSCR.FI ← (vx_flag=0) & xx_flag

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 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 63, “Actions for xscvqpswz,” on page 551.

Special Registers Altered:
- FPRF (undefined) FR (set to 0) FI FX VXSNAN VXCVI XX

VSR Data Layout for xscvqpswz

<table>
<thead>
<tr>
<th>VSR[VRB+32]</th>
</tr>
</thead>
<tbody>
<tr>
<td>src</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>VSR[VRT+32]</th>
</tr>
</thead>
<tbody>
<tr>
<td>src</td>
</tr>
</tbody>
</table>

| tgt.dword[0] | 0x0000_0000_0000_0000 |

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 +2^{31}-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 -2^{31}, 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 63, “Actions for xscvqpswz,” on page 551.

Special Registers Altered:
- FPRF (undefined) FR (set to 0) FI FX VXSNAN VXCVI XX

VSR Data Layout for xscvqpswz

<table>
<thead>
<tr>
<th>VSR[VRB+32]</th>
</tr>
</thead>
<tbody>
<tr>
<td>src</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>VSR[VRT+32]</th>
</tr>
</thead>
<tbody>
<tr>
<td>src</td>
</tr>
</tbody>
</table>

| tgt.dword[0] | 0x0000_0000_0000_0000 |
Chapter 7. Vector-Scalar Floating-Point Operations

Table 63. Actions for xscvqpswz

| src ≤ Nmin-1 | 0 – – T(Nmin), fr(0), fi(0), fprf(0bUUUUU), fx(VXCVI) |
| Nmin-1 < src ≤ Nmin | 1 – – T(Nmin), fr(0), fi(0), fx(VXCVI), error() |
| src = Nmin | – – no T(Nmin), fr(0), fi(0), fprf(0bUUUUU) |
| Nmin < src < Nmax | – – no T(bfp_CONVERT_TO_SI64(trunc(src))), fr(0), fi(0), fprf(0bUUUUU) |
| src = Nmax | – – no T(Nmax), fr(0), fi(0), fprf(0bUUUUU) |
| Nmax < src < Nmax+1 | – – no T(bfp_CONVERT_TO_SI64(trunc(src))), fr(0), fi(0), fprf(0bUUUUU), fx(XX), error() |
| src ≥ Nmax+1 | – – no T(Nmax+1), fr(0), fi(0), fprf(0bUUUUU), fx(VXCVI) |
| src is a QNaN | – – no T(Nmin), fr(0), fi(0), fprf(0bUUUUU), fx(VXCVI), fx(VXSNAN) |
| src is a SNaN | – – no T(Nmin), fr(0), fi(0), fprf(0bUUUUU), fx(VXCVI), fx(VXSNAN), error() |

Explanation:

\( T(x) \) Places the value \( x \) into the target VSR.

\[ \text{VSR}[\text{VAT}+82].\text{dword}[0] = x \]
\[ \text{VSR}[\text{VAT}+82].\text{dword}[1] = 0_{\text{XXXX}, \text{XXXX}, \text{XXXX}, \text{XXXX}} \]

\( \text{Nmin} \) The smallest signed integer word value, \(-2^{31} \times 0_{\text{FFFF}, \text{FFFF}, \text{FFFF}, \text{FFFF}}\).

\( \text{Nmax} \) The largest signed integer word value, \(2^{31}-1 \times 0_{\text{XXXX}, \text{XXXX}, \text{FFFF}, \text{FFFF}}\).

\( \text{src} \) The quad-precision floating-point value in VSR[VRB+32].

\( \text{fx}(x) \) FPSCR.FX is set to 1 if FPSCR.x=0. FPSCR.x is set to 1.

\( \text{fi}(x) \) FPSCR.FI is set to the value \( x \).

\( \text{fr}(x) \) FPSCR.FR is set to the value \( x \).

\( \text{fprf}(x) \) FPSCR.FPRF is set to the value \( x \).

\( \text{error()} \) The system error handler is invoked for the trap-enabled exception if MSR.FE0 and 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.

Explanation:

\( T(x) \) Places the value \( x \) into the target VSR.

\[ \text{VSR}[\text{VAT}+82].\text{dword}[0] = x \]
\[ \text{VSR}[\text{VAT}+82].\text{dword}[1] = 0_{\text{XXXX}, \text{XXXX}, \text{XXXX}, \text{XXXX}} \]

\( \text{Nmin} \) The smallest signed integer word value, \(-2^{31} \times 0_{\text{FFFF}, \text{FFFF}, \text{FFFF}, \text{FFFF}}\).

\( \text{Nmax} \) The largest signed integer word value, \(2^{31}-1 \times 0_{\text{XXXX}, \text{XXXX}, \text{FFFF}, \text{FFFF}}\).

\( \text{src} \) The quad-precision floating-point value in VSR[VRB+32].

\( \text{fx}(x) \) FPSCR.FX is set to 1 if FPSCR.x=0. FPSCR.x is set to 1.

\( \text{fi}(x) \) FPSCR.FI is set to the value \( x \).

\( \text{fr}(x) \) FPSCR.FR is set to the value \( x \).

\( \text{fprf}(x) \) FPSCR.FPRF is set to the value \( x \).

\( \text{error()} \) The system error handler is invoked for the trap-enabled exception if MSR.FE0 and 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.
VSX Scalar Convert with round to zero Quad-Precision to Unsigned Doubleword format X-form

\[ \text{xscvqpudz} \quad \text{VRT,VRB} \]

<table>
<thead>
<tr>
<th>63</th>
<th>VRT</th>
<th>17</th>
<th>VRB</th>
<th>836</th>
</tr>
</thead>
</table>

- If MSR.VSX=0 then VSX_Unavailable()
- reset_fflags()
- src \( \leftarrow \) bfp_CONVERT_FROM_BFP128(VSR[VRB+32])
- if src.class.NaNN | src.class.SNaN then do
  - result \( \leftarrow 0x0000_0000_0000_0000 \)
  - vxsnan_flag \( \leftarrow \) src.class.SNaN
  - vxci_flag \( \leftarrow 1 \)
- else if src.class.Infinity then do
  - vxci_flag \( \leftarrow 1 \)
  - if src.sign = 0 then
    - result \( \leftarrow 0xFFFF_FFFF_FFFF_FFFF \)
  - else
    - result \( \leftarrow 0x0000_0000_0000_0000 \)
- else if src.class.Zero then result \( \leftarrow 0x0000_0000_0000_0000 \)
  - else do
    - rnd \( \leftarrow \) bfp_ROUND_TO_INTEGER(O0001,src)
    - if bfp_COMPARE_GT(rnd, +264-1) then do
      - result \( \leftarrow 0xFFFF_FFFF_FFFF_FFFF \)
      - vxci_flag \( \leftarrow 1 \)
    - else if bfp_COMPARE_LT(rnd, 0) then do
      - result \( \leftarrow 0x0000_0000_0000_0000 \)
      - vxci_flag \( \leftarrow 1 \)
    - else do
      - result \( \leftarrow \) bfp_CONVERT_TO_UI64(rnd)
      - if(xx_flag) then SetFX(FPSCR.XX)
  - end
  - if(vxsnan_flag) then SetFX(FPSCR.VXSNAN)
  - if(vxci_flag) then SetFX(FPSCR.VXCVI)
- vx_flag \( \leftarrow \) vxsnan_flag | vxci_flag
  - vxci_flag \( \leftarrow \) FPSCR.VXCVI & vx_flag
  - if |vx_flag|=0 then do
    - VSR[VRT+32].dword[0] \( \leftarrow \) result
  - VSR[VRT+32].dword[1] \( \leftarrow 0x0000_0000_0000_0000 \)
  - FPSCR.FPRF \( \leftarrow 0bUUUUU \)
  - FPSCR.FR \( \leftarrow \) vx_flag & inc_flag
  - FPSCR.FI \( \leftarrow \) vx_flag & xx_flag

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 NaN, 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 64, “Actions for xscvqpudz,” on page 553.

Special Registers Altered:
- FPRF (undefined) FR (set to 0) FI FX VXSNAN VXCVI XX

VSR Data Layout for xscvqpudz

<table>
<thead>
<tr>
<th>VSR[VRB+32]</th>
<th>src</th>
</tr>
</thead>
<tbody>
<tr>
<td>VSR[VRT+32]</td>
<td>tgt.dword[0]</td>
</tr>
</tbody>
</table>
### Returned Results and Status Setting

<table>
<thead>
<tr>
<th>src ≤ Nmin-1</th>
<th>0</th>
<th>–</th>
<th>–</th>
<th>T(Nmin), fr(0), fi(0), fprf(0bUUUUU), fx(VXCVI)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>–</td>
<td>–</td>
<td>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>0</th>
<th>yes</th>
<th>T(Nmin), fr(0), fi(1), fprf(0bUUUUU), fx(KX), error()</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>–</td>
<td>1</td>
<td>yes</td>
<td>T(Nmin), fr(0), fi(1), fprf(0bUUUUU), fx(KX), error()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>src = Nmin</th>
<th>–</th>
<th>–</th>
<th>no</th>
<th>T(Nmin), fr(0), fi(0), fprf(0bUUUUU)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nmin &lt; src &lt; Nmax</td>
<td>–</td>
<td>0</td>
<td>yes</td>
<td>T(Nmax), fr(0), fi(1), fprf(0bUUUUU), fx(KX)</td>
</tr>
<tr>
<td></td>
<td>–</td>
<td>1</td>
<td>yes</td>
<td>T(Nmax), fr(0), fi(1), fprf(0bUUUUU), fx(KX), error()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>src ≥ Nmax</th>
<th>0</th>
<th>–</th>
<th>–</th>
<th>T(Nmax), fr(0), fi(0), fprf(0bUUUUU), fx(VXCVI)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>–</td>
<td>–</td>
<td>fr(0), fi(0), fx(VXCVI), error()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>src is a QNaN</th>
<th>0</th>
<th>–</th>
<th>–</th>
<th>T(Nmin), fr(0), fi(0), fprf(0bUUUUU), fx(VXCVI), fx(VXSNAN)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>–</td>
<td>–</td>
<td>fr(0), fi(0), fx(VXCVI), fx(VXSNAN), error()</td>
</tr>
</tbody>
</table>

### Explanation:

- \( T(x) \) places the value \( x \) into the target VSR.
- \( VSR[VRB+32]: \text{dword}[1] \leftarrow x \)
- \( VSR[VRB+32]: \text{dword}[0] \leftarrow 0x0000_0000_0000_0000 \)
- \( Nmin \) is the smallest unsigned integer doubleword value, \( 0 (0x0000_0000_0000_0000) \).
- \( Nmax \) is the largest unsigned integer doubleword value, \( 2^{64}-1 (0xFFFF_FFFF_FFFF_FFFF) \).
- \( src \) is the quad-precision floating-point value in \( VSR[VRB+32] \).
- \( fx(x) \) is set to 1 if FPSCR.FX is set to 1.
- \( fi(x) \) is set to the value \( x \).
- \( fr(x) \) is set to the value \( x \).
- \( fprf(x) \) is set to the value \( x \).
- \( error() \) is set to 1 if MSR.FE0 and MSR.FE1 are set to any mode other than the ignore-exception mode.

- \( trunc(x) \) returns the floating-point value \( x \) truncated to a floating-point integer.

| Table 64. Actions for xscvpqdz |
VSX Scalar Convert with round to zero
Quad-Precision to Unsigned Word format
X-form

xscvqpuwz    VRT,VRB

<table>
<thead>
<tr>
<th></th>
<th>VRT</th>
<th></th>
<th></th>
<th>VRB</th>
<th>836</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td>6</td>
<td>1</td>
<td>16</td>
<td>21</td>
<td></td>
</tr>
</tbody>
</table>

if MSR.VSX=0 then VSX_Unavailable()
reset staffers()

src ← bfp_CONVERT_FROM_BFP128(VSR[VRB+32])

if src.class.QNaN | src.class.SNaN then do
result ← 0x0000_0000
vxsnan_flag ← src.class.SNaN
vxcvi_flag ← 1
end
else if src.class.Infinity then do
vxcvi_flag ← 1
if src.sign = 0 then
result ← 0x0000_0000_FFFF_FFFF
else
result ← 0x0000_0000_0000_0000
end
else if src.class.Zero then
result ← 0x0000_0000
else do
rnd ← bfp_ROUND_TO_INTEGER(0b001,src)
if bfp_COMPARE_GT(rnd, +2^{32}-1) then do
result ← 0x0000_0000_FFFF_FFFF
vxcvi_flag ← 1
end
else if bfp_COMPARE_LT(rnd, bfp_ZERO) then do
result ← 0x0000_0000_0000_0000
vxcvi_flag ← 1
end
else do
result ← bfp_CONVERT_TO_UI64(rnd)
if(xx_flag) then SetFX(FPSCR.XX)
end
endif(vxsnan_flag) then SetFX(FPSCR.VXSNAN)
if(vxcvi_flag)  then SetFX(FPSCR.VXCVI)
.vx_flag ← vxsnan_flag | vxcvi_flag
.ex_flag ← FPSCR.XE & vx_flag

if ex_flag=0 then do
VSR[VRT+32].dword[0] ← result
VSR[VRT+32].dword[1] ← 0x0000_0000_0000_0000
FPSCR.FR ← 0bUUUUU
end
FPSCR.FR ← (vx_flag=0) & inc_flag
FPSCR.FI ← (vx_flag=0) & xx_flag

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 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 0x0000_0000_FFFF_FFFF.

Otherwise, do the following.
Let rnd be the value src truncated to a floating-point integer.

If rnd is greater than +2^{32}-1, an Invalid Operation exception occurs, VXCVI is set to 1, and the result is 0x0000_0000_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 65, “Actions for xscvqpuwz,” on page 555.

Special Registers Altered:
FPRF (undefined) FR (set to 0) FI FX VXSNAN VXCVI XX

VSR Data Layout for xscvqpuwz

<table>
<thead>
<tr>
<th>src</th>
</tr>
</thead>
</table>

| tgt.dword[0] | 0x0000_0000_0000_0000 |
### Returned Results and Status Setting

<table>
<thead>
<tr>
<th>src ≤ Nmin-1</th>
<th>0</th>
<th>–</th>
<th>T(Nmin), fr(0), fi(0), fprf(0bUUUUU), fx(VXCVI)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>–</td>
<td>–</td>
<td>fr(0), fi(0), fx(VXCVI), error()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Nmin-1 &lt; src &lt; Nmin</th>
<th>0</th>
<th>yes</th>
<th>T(Nmin), fr(0), fi(1), fprf(0bUUUUU), fx(XX), error()</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>yes</td>
<td>T(Nmin), fr(0), fi(1), fprf(0bUUUUU), fx(XX), error()</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>src = Nmin</th>
<th>0</th>
<th>–</th>
<th>T(Nmin), fr(0), fi(0), fprf(0bUUUUU)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>–</td>
<td>no</td>
<td>T(Nmin), fr(0), fi(0), fprf(0bUUUUU), fx(VXCVI)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Nmin &lt; src &lt; Nmax</th>
<th>–</th>
<th>yes</th>
<th>T(bfp_CONVERT_TO_UI64(trunc(src))), fr(0), fi(1), fprf(0bUUUUU), fx(XX)</th>
</tr>
</thead>
<tbody>
<tr>
<td>–</td>
<td>yes</td>
<td>T(bfp_CONVERT_TO_UI64(trunc(src))), fr(0), fi(1), fprf(0bUUUUU), fx(XX), error()</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>src = Nmax</th>
<th>–</th>
<th>–</th>
<th>T(Nmax), fr(0), fi(0), fprf(0bUUUUU)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>–</td>
<td>no</td>
<td>T(Nmax), fr(0), fi(0), fprf(0bUUUUU), fx(VXCVI)</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), fprf(0bUUUUU), fx(XX)</th>
</tr>
</thead>
<tbody>
<tr>
<td>–</td>
<td>yes</td>
<td>T(Nmax), fr(0), fi(0), fprf(0bUUUUU), fx(XX), error()</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>src ≥ Nmax+1</th>
<th>0</th>
<th>–</th>
<th>T(Nmax), fr(0), fi(0), fprf(0bUUUUU), fx(VXCVI)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>–</td>
<td>fr(0), fi(0), fx(VXCVI), error()</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>src is a QNaN</th>
<th>0</th>
<th>–</th>
<th>T(Nmin), fr(0), fi(0), fprf(0bUUUUU), fx(VXCVI)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>–</td>
<td>fr(0), fi(0), fx(VXCVI), fx(VXSNAN), error()</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>src is a SNaN</th>
<th>0</th>
<th>–</th>
<th>T(Nmin), fr(0), fi(0), fprf(0bUUUUU), fx(VXCVI), fx(VXSNAN)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>–</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.
  
  \[
  \begin{align*}
  \text{VSR}[\text{VAT+02}].\text{dword}[0] & = x \\
  \text{VSR}[\text{VAT+02}].\text{dword}[1] & = 0x0000_0000_0000_0000
  \end{align*}
  \]

- **Nmin** The smallest unsigned integer word value, \( 0 (0x0000_0000_0000_0000) \).

- **Nmax** The largest unsigned integer word value, \( 2^{32}-1 (0x0000_0000_0000_0000) \).

- **src** The quad-precision floating-point value in VSR[VRT+32].

- **fx(x)** FPSCR.FX is set to 1 if FPSCR.x=0. 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 \( x \).

- **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 65. Actions for xscvqpuwz

---

**Chapter 7. Vector-Scalar Floating-Point Operations**

555
VSX Scalar Convert Signed Doubleword to Quad-Precision format X-form

xscvsdq

<table>
<thead>
<tr>
<th>63</th>
<th>35</th>
<th>86</th>
<th>21</th>
<th>236</th>
</tr>
</thead>
<tbody>
<tr>
<td>VRT</td>
<td>VRB</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Let \( \text{src} \) be the signed integer value in doubleword element 0 of VSR[VRB+32].

\( \text{src} \) is placed into VSR[VRT+32] in quad-precision floating-point format.

FPFR is set to the class and sign of the result. FR is set to 0. FI is set to 0.

Special Registers Altered:

FPFR FR (set to 0) FI (set to 0)

VSR Data Layout for xscvsdq

<table>
<thead>
<tr>
<th>VSR[VRB+32]</th>
</tr>
</thead>
<tbody>
<tr>
<td>src.dword[0]</td>
</tr>
<tr>
<td>VSR[VRT+32]</td>
</tr>
<tr>
<td>tgt</td>
</tr>
</tbody>
</table>
VSX Scalar Convert Single-Precision to Double-Precision format XX2-form

xscvspdp XT,XB

<table>
<thead>
<tr>
<th></th>
<th>T</th>
<th>///</th>
<th>B</th>
<th>329</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>
</tr>
</tbody>
</table>

reset_xflags();

src ← VSR[32×BX+B].word[0]
result ← ConvertVector9ToScalar5P(src)
if (vxsnan_flag) then SetFX(FPSCR.VXSNAN)
veq_flag ← FPSCR.VE & vxsnan_flag
FPSCR.FR ← 000
FPSCR.FI ← 000
if (~veq_flag) then do
VSR[32×TX+T].dword[0] ← result
VSR[32×TX+T].dword[1] ← 0xUUUU_UUUU_UUUU_UUUU
FPSCR.FPRF ← ClassDP(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 undefined.

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

src = VSR[XB]

/help/undefined
unused

tgt = VSR[XT]

/help/undefined

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.
VSX Scalar Convert Single-Precision to Double-Precision format Non-signalling XX2-form

xscvspdpn XT,XB

<table>
<thead>
<tr>
<th></th>
<th>60</th>
<th>T</th>
<th>B</th>
<th>331</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>
</tr>
</tbody>
</table>

reset_xflags();
src ← VSR[32×BX+B].word[0];
result ← ConvertSPtoDP_NS(src);
VSR[32×TX+T].dword[0] ← result;
VSR[32×TX+T].dword[1] ← 0xUUUU_UUUU_UUUU_UUUU

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 undefined.

Special Registers Altered
None

VSR Data Layout for xscvspdpn

```plaintext
crc = VSR[XB]

<table>
<thead>
<tr>
<th>.word[0]</th>
<th>unused</th>
<th>unused</th>
<th>unused</th>
</tr>
</thead>
</table>

tgt = VSR[XT]

<table>
<thead>
<tr>
<th>.dword[0]</th>
<th>undefined</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>32</td>
</tr>
</tbody>
</table>
```

Programming Note

xscvsdp 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.
VSX Scalar Convert with round Signed Doubleword to Double-Precision format XX2-form

\[ \text{xscvssdpp} \quad \text{XT, XB} \]

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>376</th>
<th>31x</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>312</th>
<th>31x</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\begin{align*}
\text{reset}_xflagn() & \leftarrow \text{ConvertSDtoFP}([VSR[32\times X + B].\text{wword}[0]]) \leftarrow \text{RoundToDP}(\text{RN}, src) \\
\text{VSR}[32\times X + T].\text{wword}[0] & \leftarrow 0_{\text{UUUU}_U\text{UUUU}_U\text{UUUU}_U\text{UUUU}} \\
\text{if}(\text{xx}_\text{flag}) & \text{then SetFX}(XX) \\
\text{FPRF} & \leftarrow \text{ClassDP}(result) \leftarrow \text{inc}_\text{flag} \\
\text{FI} & \leftarrow \text{xx}_\text{flag} \\
\text{Let XT be the value } 32\times T + T. \\
\text{Let XB be the value } 32\times B + B. \\
\text{Let } src \text{ be the signed integer value in doubleword element 0 of VSR[XB].} \\
\text{src} \text{ 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 undefined. \\
\text{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.} \\
\text{Special Registers Altered} \quad \text{FPRF FR FI FX XX}
\end{align*}

VSX Scalar Convert with round Signed Doubleword to Single-Precision format XX2-form

\[ \text{xscvssdsp} \quad \text{XT, XB} \]

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>312</th>
<th>31x</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>312</th>
<th>31x</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\begin{align*}
\text{reset}_xflagn() & \leftarrow \text{ConvertSDtoFP}([VSR[32\times X + B].\text{wword}[0]]) \leftarrow \text{RoundToSP}(\text{RN}, src) \\
\text{VSR}[32\times X + T].\text{wword}[0] & \leftarrow \text{ConvertSPtoSP64}(\text{result}) \leftarrow 0_{\text{UUUU}_U\text{UUUU}_U\text{UUUU}_U\text{UUUU}} \\
\text{if}(\text{xx}_\text{flag}) & \text{then SetFX}(XX) \\
\text{FPRF} & \leftarrow \text{ClassSP}(result) \leftarrow \text{inc}_\text{flag} \\
\text{FI} & \leftarrow \text{xx}_\text{flag} \\
\text{Let XT be the value } 32\times T + T. \\
\text{Let XB be the value } 32\times B + B. \\
\text{Let } src \text{ be the two's-complement integer value in doubleword element 0 of VSR[XB].} \\
\text{src} \text{ 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 undefined. \\
\text{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.} \\
\text{Special Registers Altered} \quad \text{FPRF FR FI FX XX}
\end{align*}

VSR Data Layout for xscvssdpp

\begin{itemize}
\item src = VSR[XB] \\
\item tgt = VSR[XT]
\end{itemize}

\begin{itemize}
\item SD | unused |
\item DP | undefined |
\end{itemize}

VSR Data Layout for xscvssdsp

\begin{itemize}
\item src = VSR[XB] \\
\item tgt = VSR[XT]
\end{itemize}

\begin{itemize}
\item SD | unused |
\item DP | undefined |
\end{itemize}
**VSX Scalar Convert Signed Doubleword to Quad-Precision format X-form**

`xscvsdqp` VRT,VRB

<table>
<thead>
<tr>
<th>src</th>
<th>tgt</th>
<th>VSR[VRT+32]</th>
<th>VSR[VRB+32]</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td>10</td>
<td>11</td>
<td>16</td>
</tr>
<tr>
<td>836</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

if MSR.VSX=0 then VSX_Unavailable()

Let `src` be the signed integer value in doubleword element 0 of VSR[VRB+32].

`src` is placed into VSR[VRT+32] in quad-precision floating-point format.

FPFR is set to the class and sign of the result. FR is set to 0. FI is set to 0.

**Special Registers Altered:**

FPFR FR (set to 0) FI (set to 0)

**VSR Data Layout for xscvsdqp**

<table>
<thead>
<tr>
<th>VSR[VRB+32]</th>
<th>VSR[VRT+32]</th>
</tr>
</thead>
<tbody>
<tr>
<td>src.dword[0]</td>
<td>unused</td>
</tr>
<tr>
<td>tgt</td>
<td></td>
</tr>
</tbody>
</table>

**VSX Scalar Convert Unsigned Doubleword to Quad-Precision format X-form**

`xscvudqp` VRT,VRB

<table>
<thead>
<tr>
<th>src</th>
<th>tgt</th>
<th>VSR[VRT+32]</th>
<th>VSR[VRB+32]</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td>10</td>
<td>11</td>
<td>16</td>
</tr>
<tr>
<td>836</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

if MSR.VSX=0 then VSX_Unavailable()

Let `src` be the unsigned integer value in doubleword element 0 of VSR[VRB+32].

`src` is placed into VSR[VRT+32] in quad-precision floating-point format.

FPFR is set to the class and sign of the result. FR is set to 0. FI is set to 0.

**Special Registers Altered:**

FPFR FR (set to 0) FI (set to 0)

**VSR Data Layout for xscvudqp**

<table>
<thead>
<tr>
<th>VSR[VRB+32]</th>
<th>VSR[VRT+32]</th>
</tr>
</thead>
<tbody>
<tr>
<td>src.dword[0]</td>
<td>unused</td>
</tr>
<tr>
<td>tgt</td>
<td></td>
</tr>
</tbody>
</table>

---

63 VRT 10 VRB 836

/ 0 6 11 16 21 31

if MSR.VSX=0 then VSX_Unavailable()

src ← bfp_CONVERT_FROM_SI64(VSR[VRB+32].dword[0])

result ← bfp_CONVERT_TO_BFP128(src)

VSR[VRT+32] ← result

FPSCR.FPRF ← fprf_CLASS_BFP128(result)

FPSCR.FR ← 0

FPSCR.FI ← 0

63 VRT 2 VRB 836

/ 0 6 11 16 21 31

if MSR.VSX=0 then VSX_Unavailable()

src ← bfp_CONVERT_FROM_UI64(VSR[VRB+32].dword[0])

result ← bfp_CONVERT_TO_BFP128(src)

VSR[VRT+32] ← result

FPSCR.FPRF ← fprf_CLASS_BFP128(result)

FPSCR.FR ← 0

FPSCR.FI ← 0
VSX Scalar Convert with round Unsigned Doubleword to Double-Precision format

`xscvuxddp` XT,XB

<table>
<thead>
<tr>
<th>60</th>
<th>T</th>
<th>/</th>
<th>B</th>
<th>360</th>
<th>81X</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>

reset_xflags();

src ← ConvertUDtoFP(VSR[32×BX+B].dword[0]);
result ← RoundToDP(RN,src);
VSR[32×TX+T].dword[0] ← result
VSR[32×TX+T].dword[1] ← 0xUUUU_UUUU_UUUU_UUUU

if(xx_flag) then SetFX(XX)

FPRF ← ClassDP(result);
FR ← inc_flag;
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 undefined.

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 xscvuxddp**

src = VSR[XB]

<table>
<thead>
<tr>
<th>UD</th>
<th>unused</th>
</tr>
</thead>
</table>
tgt = VSR[XT]

<table>
<thead>
<tr>
<th>DP</th>
<th>undefined</th>
</tr>
</thead>
</table>

VSX Scalar Convert with round Unsigned Doubleword to Single-Precision XX2-form

`xscvuxdsp` XT,XB

<table>
<thead>
<tr>
<th>60</th>
<th>T</th>
<th>///</th>
<th>B</th>
<th>296</th>
<th>b7X</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>

reset_xflags();

src ← ConvertUDtoFP(VSR[32×BX+B].dword[0]);
result ← RoundToSP(RN,src);
VSR[32×TX+T].dword[0] ← ConvertSPtoSP64(result);
VSR[32×TX+T].dword[1] ← 0xUUUU_UUUU_UUUU_UUUU

if(xx_flag) then SetFX(XX)

FPRF ← ClassSP(result);
FR ← inc_flag;
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 undefined.

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.

**Special Registers Altered**

FPRF FR FI FX XX

**VSR Data Layout for xscvuxdsp**

src = VSR[XB]

<table>
<thead>
<tr>
<th>UD</th>
<th>unused</th>
</tr>
</thead>
</table>
tgt = VSR[XT]

<table>
<thead>
<tr>
<th>DP</th>
<th>undefined</th>
</tr>
</thead>
</table>

0 64 127
VSX Scalar Divide Double-Precision XX3-form

**xsdivdp**  \( \text{XT}, \text{XA}, \text{XB} \)

<table>
<thead>
<tr>
<th>0</th>
<th>60</th>
<th>6</th>
<th>51</th>
<th>46</th>
<th>41</th>
<th>36</th>
<th>31</th>
<th>26</th>
<th>21</th>
<th>16</th>
<th>11</th>
<th>6</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>XT</td>
<td>+ TX</td>
<td></td>
<td>T</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>XA</td>
<td>+ AX</td>
<td></td>
<td>A</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>XB</td>
<td>+ BX</td>
<td></td>
<td>B</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>reset_flags()</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>src1</td>
<td>+ VSR[XA][0:63]</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>src2</td>
<td>+ VSR[XB][0:63]</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>v(0:1:63)</td>
<td>= DivideFP(src1, src2)</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(0:63)</td>
<td>= RoundToFPRN(v, v)</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>if(vxssnan_flag) then SetFX(VXSSNAN)</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>if(vxidi_flag) then SetFX(VXIDI)</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>if(vxzdz_flag) then SetFX(VXZDZ)</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>if(ox_flag) then SetFX(OX)</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>if(ux_flag) then SetFX(UX)</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>if(xx_flag) then SetFX(XX)</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>if(zx_flag) then SetFX(ZX)</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>vex_flag</td>
<td>= VE &amp; (vxssnan_flag</td>
<td>vxidi_flag</td>
<td>vxzdz_flag)</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>zex_flag</td>
<td>= ZE &amp; zx_flag</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>if(~vex_flag &amp; ~zex_flag) then do</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>VSR[XT] = result</td>
<td></td>
<td>0xUUUU_UUUU_UUUU_UUUU</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>FPRF = ClassDP(result)</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>FR = inc_flag</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>FI = xx_flag</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>end</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>else do</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>FR = 0b0</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>FI = 0b0</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>end</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>

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].

Let src2 be the double-precision floating-point value in doubleword element 0 of VSR[XB].

src1 is divided\(^1\) by src2, producing a quotient having unbounded range and precision.

The quotient is normalized\(^2\).

See **Actions for xsdivdp** (p. 563).

The intermediate result is rounded to double-precision using the rounding mode specified by RN.

See **Table 50, “Scalar Floating-Point Intermediate Result Handling,”** on page 515.

---

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 66. 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 v ← +Infinity  v ← +Infinity  v ← -Infinity  v ← dQNaN v ← +Infinity v ← -Infinity v ← dQNaN v ← +Infinity  v ← src2 v ← Q(src2) vxidi_flag 1 vxidi_flag 1 vxidi_flag 1 vxidi_flag 1 vxidi_flag 1 vxidi_flag 1 vxidi_flag 1 vxidi_flag 1 vxidi_flag 1 vxidi_flag 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-NZF</td>
<td>v ← +Zero  v ← D(src1, src2) v ← +Infinity  v ← -Infinity  v ← D(src1, src2) v ← -Zero  v ← src2 v ← Q(src2) v ← src2 v ← Q(src2) vxidi_flag 1 vxidi_flag 1 vxidi_flag 1 vxidi_flag 1 vxidi_flag 1 vxidi_flag 1 vxidi_flag 1 vxidi_flag 1 vxidi_flag 1 vxidi_flag 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-Zero</td>
<td>v ← +Zero  v ← +Zero  v ← dQNaN v ← +Infinity  v ← dQNaN v ← -Zero  v ← src2 v ← Q(src2) v ← src2 v ← Q(src2) vxidi_flag 1 vxidi_flag 1 vxidi_flag 1 vxidi_flag 1 vxidi_flag 1 vxidi_flag 1 vxidi_flag 1 vxidi_flag 1 vxidi_flag 1 vxidi_flag 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+Zero</td>
<td>v ← -Zero  v ← -Zero  v ← dQNaN v ← -Infinity  v ← dQNaN v ← +Zero  v ← src2 v ← Q(src2) v ← src2 v ← Q(src2) vxidi_flag 1 vxidi_flag 1 vxidi_flag 1 vxidi_flag 1 vxidi_flag 1 vxidi_flag 1 vxidi_flag 1 vxidi_flag 1 vxidi_flag 1 vxidi_flag 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+NZF</td>
<td>v ← -Zero  v ← D(src1, src2) v ← -Infinity  v ← +Infinity  v ← D(src1, src2) v ← +Zero  v ← src2 v ← Q(src2) v ← src2 v ← Q(src2) vxidi_flag 1 vxidi_flag 1 vxidi_flag 1 vxidi_flag 1 vxidi_flag 1 vxidi_flag 1 vxidi_flag 1 vxidi_flag 1 vxidi_flag 1 vxidi_flag 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+Infinity</td>
<td>v ← dQNaN v ← +Infinity  v ← +Infinity  v ← -Infinity  v ← dQNaN v ← +Infinity v ← +Infinity v ← dQNaN v ← +Infinity  v ← src2 v ← Q(src2) vxidi_flag 1 vxidi_flag 1 vxidi_flag 1 vxidi_flag 1 vxidi_flag 1 vxidi_flag 1 vxidi_flag 1 vxidi_flag 1 vxidi_flag 1 vxidi_flag 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>QNaN</td>
<td>v ← src1  v ← src1  v ← src1  v ← src1  v ← src1  v ← src1  v ← src1  v ← src1  v ← src1  v ← src1  vxsnan_flag 1 vxsnan_flag 1 vxsnan_flag 1 vxsnan_flag 1 vxsnan_flag 1 vxsnan_flag 1 vxsnan_flag 1 vxsnan_flag 1 vxsnan_flag 1 vxsnan_flag 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SNaN</td>
<td>v ← Q(src1) vxsnan_flag 1 v ← Q(src1) vxsnan_flag 1 v ← Q(src1) vxsnan_flag 1 v ← Q(src1) vxsnan_flag 1 v ← Q(src1) vxsnan_flag 1 v ← Q(src1) vxsnan_flag 1 v ← Q(src1) vxsnan_flag 1 v ← Q(src1) vxsnan_flag 1 vxsnan_flag 1 vxsnan_flag 1 vxsnan_flag 1 vxsnan_flag 1 vxsnan_flag 1 vxsnan_flag 1 vxsnan_flag 1 vxsnan_flag 1 vxsnan_flag 1 vxsnan_flag 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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 VXDI 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 Invalid Divide exception occurs and VXZDZ 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 src2.

Otherwise, if src1 and src2 are Infinity values, or if src1 and src2 are Zero values, the result is the default Quiet NaN1.

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 67, “Actions for xsdivqp[o],” on page 565.

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.

See Table 50, “Scalar Floating-Point Intermediate Result Handling,” on page 515.

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 51, “VSX Scalar Floating-Point Final Result,” on page 516.

Special Registers Altered:

FPRF FR FI FX VXSNAN VXDI VXZDZ OX UX ZX XX

1. The quad-precision default Quiet NaN is the value, 0x7FFF_8000_0000_0000_0000_0000_0000.
### VSR Data Layout for xsdivqp[o]

<table>
<thead>
<tr>
<th>src1</th>
<th>src2</th>
<th>tgt</th>
</tr>
</thead>
<tbody>
<tr>
<td>src1</td>
<td>src2</td>
<td>tgt</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]`.
- **dQNaN**: Default quiet NaN (`x'7FFF_8000_0000_0000_0000_0000_0000`).
- **NZF**: Nonzero finite number.
- **Div(x,y)**: The floating-point value `x` is divided by `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 67. Actions for xsdivqp[o]

<table>
<thead>
<tr>
<th>Operation</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>QNaN</th>
<th>SNaN</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 + dQNaN</td>
<td>v + infinity</td>
<td>v + q NaN</td>
<td>v + q NaN</td>
</tr>
<tr>
<td><code>-NZF</code></td>
<td>v + Div(src1, src2)</td>
<td>v + infinity</td>
<td>v - infinity</td>
<td>v + infinity</td>
<td>v + Div(src1, src2)</td>
<td>v + q NaN</td>
<td>v + q NaN</td>
<td></td>
</tr>
<tr>
<td><code>-Zero</code></td>
<td>v + Zero</td>
<td>v + dQNaN</td>
<td>v + infinity</td>
<td>v + infinity</td>
<td>v + dQNaN</td>
<td>v + infinity</td>
<td>v + q NaN</td>
<td>v + q NaN</td>
</tr>
<tr>
<td><code>+Zero</code></td>
<td>v - Zero</td>
<td>v + dQNaN</td>
<td>v + infinity</td>
<td>v + infinity</td>
<td>v + dQNaN</td>
<td>v + infinity</td>
<td>v + q NaN</td>
<td>v + q NaN</td>
</tr>
<tr>
<td><code>+NZF</code></td>
<td>v + Div(src1, src2)</td>
<td>v + infinity</td>
<td>v - infinity</td>
<td>v + infinity</td>
<td>v + Div(src1, src2)</td>
<td>v + q NaN</td>
<td>v + q NaN</td>
<td></td>
</tr>
<tr>
<td><code>+Infinity</code></td>
<td>v + dQNaN</td>
<td>v + infinity</td>
<td>v - infinity</td>
<td>v + infinity</td>
<td>v + dQNaN</td>
<td>v + infinity</td>
<td>v + q NaN</td>
<td>v + q NaN</td>
</tr>
<tr>
<td>QNaN</td>
<td>v + q NaN</td>
<td>v + q NaN</td>
<td>v + q NaN</td>
<td>v + q NaN</td>
<td>v + q NaN</td>
<td>v + q NaN</td>
<td>v + q NaN</td>
<td>v + q NaN</td>
</tr>
<tr>
<td>Snan</td>
<td>v + q NaN</td>
<td>v + q NaN</td>
<td>v + q NaN</td>
<td>v + q NaN</td>
<td>v + q NaN</td>
<td>v + q NaN</td>
<td>v + q NaN</td>
<td>v + q NaN</td>
</tr>
</tbody>
</table>

#### Notes:
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

\texttt{xsdvsp} \ XT,XA,XB

\begin{center}
\begin{tabular}{cccccc}
\texttt{X} & \texttt{T} & \texttt{A} & \texttt{B} & \texttt{24}
\hline
0 & 6 & 11 & 16 & 21 & 24
\end{tabular}
\end{center}

reset_xflags()

\begin{Verbatim}
src1 \leftarrow \text{VSR}[32 \times \text{XA} + \text{A}].\text{dword}[0]
src2 \leftarrow \text{VSR}[32 \times \text{XB} + \text{B}].\text{dword}[0]
v \leftarrow \text{DivideDP}(\text{src1}, \text{src2})
result \leftarrow \text{RoundToSP}(\text{RN}, v)
\end{Verbatim}

\begin{Verbatim}
if(\text{vxsnan_flag}) \text{then SetFX(VXSNAN)}
if(\text{vxidi_flag}) \text{then SetFX(VXIDI)}
if(\text{vxzdz_flag}) \text{then SetFX(VXZDZ)}
if(\text{xx_flag}) \text{then SetFX(XX)}
if(\text{zx_flag}) \text{then SetFX(ZX)}
\end{Verbatim}

\begin{Verbatim}
vex_flag \leftarrow \text{VE} \& (\text{vxsnan_flag} | \text{vxidi_flag} | \text{vxzdz_flag})
zex_flag \leftarrow \text{ZE} \& \text{zx_flag}
\end{Verbatim}

\begin{Verbatim}
if(\neg v\text{ex_flag} \& \neg z\text{ex_flag}) \text{then do}
\text{VSR}[32 \times \text{TX} + \text{T}].\text{dword}[0] \leftarrow \text{ConvertSPtoSP64}(\text{result})
\text{VSR}[32 \times \text{TX} + \text{T}].\text{dword}[1] \leftarrow 0xUUUU_UUUU_UUUU_UUUU
\text{FPRF} \leftarrow \text{ClassSP}(\text{result})
\text{FR} \leftarrow \text{inc_flag}
\text{FI} \leftarrow \text{xx_flag}
\text{end}
\end{Verbatim}

\begin{Verbatim}
else do
\text{FR} \leftarrow 0b0
\text{FI} \leftarrow 0b0
\text{end}
\end{Verbatim}

Let XT be the value $32 \times \text{TX} + \text{T}$.
Let XA be the value $32 \times \text{AX} + \text{A}$.
Let XB be the value $32 \times \text{BX} + \text{B}$.

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 divided\(^1\) by src2, producing a quotient having
unbounded range and precision.

The quotient is normalized\(^2\).

See Table 68, “Actions for xsdvsp,” on page 567.

The intermediate result is rounded to single-precision
using the rounding mode specified by RN.

See Table 50, “Scalar Floating-Point Intermediate
Result Handling,” on page 515.

---

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.
### Chapter 7. Vector-Scalar Floating-Point Operations

#### Table 68. Actions for `xsdivsp`

<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>v ← dQNaN vxd_flag ← 1</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← dQNaN vxd_flag ← 1</td>
<td>v ← src2</td>
<td>v ← Q(src2) vxsnan_flag ← 1</td>
<td></td>
</tr>
<tr>
<td>v ← +Zero</td>
<td>v ← D(src1,src2)</td>
<td>v ← +Infinity vxzdz_flag ← 1</td>
<td>v ← -Infinity vxzdz_flag ← 1</td>
<td>v ← D(src1,src2)</td>
<td>v ← +Zero</td>
<td>v ← +Zero</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 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 Insert Exponent Double-Precision X-form

xsiexpdp XT,RA,RB

Let $XT$ be the sum $32 \times TX + T$.

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 undefined.

Special Registers Altered:
None

Programming Note
This instruction can be used to produce a single-precision result.

VSR Data Layout for xsiexpdp

<table>
<thead>
<tr>
<th>src1</th>
<th>GPR[RA]</th>
</tr>
</thead>
<tbody>
<tr>
<td>src2</td>
<td>GPR[RB]</td>
</tr>
<tr>
<td>tgt</td>
<td>VSR[$XT$].dword[0]</td>
</tr>
</tbody>
</table>

if MSR.VSX=0 then VSX_Unavailable()
**VSX Scalar Insert Exponent Quad-Precision X-form**

**xsiexpqp**  
VRT, VRA, VRB

<table>
<thead>
<tr>
<th></th>
<th>63</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>868</th>
</tr>
</thead>
</table>

`if MSR.VSX=0 then VSX_Unavailable()`

`VSR[VRT+32].bit[0] <- VSR[VRA+32].bit[0]`

The contents of bit 0 of `VSR[VRA+32]` are placed into bit 0 of `VSR[VRT+32]`.


**Special Registers Altered:**
None

**VSR Data Layout for xsiexpqp**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>VSR[VRA+32]</code></td>
<td><code>src1</code></td>
</tr>
<tr>
<td><code>VSR[VRB+32]</code></td>
<td><code>src2.dword[0] unused</code></td>
</tr>
<tr>
<td><code>VSR[VRT+32]</code></td>
<td><code>tgt</code></td>
</tr>
</tbody>
</table>
**VSX Scalar Multiply-Add Double-Precision XX3-form**

For `xsmaddadp`, 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 69.

`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 69.

The intermediate result is rounded to double-precision using the rounding mode specified by `RN`.

See Table 50, "Scalar Floating-Point Intermediate Result Handling," on page 515.

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 undefined.

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 51, "VSX Scalar Floating-Point Final Result," on page 516.

**Special Registers Altered**

\[
\begin{align*}
&\text{FPRF, FR, FI, OX, UX, XX} \\
&VXSNAN, VXISI, VXIMZ
\end{align*}
\]

---

**Notes:**
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 `xsmadd(a|m)dp`

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>src1</td>
<td>src2</td>
<td>src3</td>
<td>tgt</td>
</tr>
<tr>
<td>DP</td>
<td>unused</td>
<td>unused</td>
<td>unused</td>
</tr>
<tr>
<td>0</td>
<td>64</td>
<td>127</td>
<td>undefined</td>
</tr>
</tbody>
</table>
### Table 69. Actions for xsmadd(a|m)dp

| src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 | src2 | src1 | src3 |

| 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 xsmaddadp, 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 Single-Precision
XX3-form

\[ xsmaddasp \quad XT,XA,XB \]

\[ \begin{array}{cccccc}
0 & 6 & 11 & 16 & 21 & 9 \\
\end{array} \]

\[ \begin{array}{cccccc}
0 & 6 & 11 & 16 & 21 & 9 \\
\end{array} \]

For \texttt{xsmaddasp}, 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 \texttt{xsmaddmsp}, 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 70, “Actions for xsmadd(a|m)sp,” on page 575.

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 70, “Actions for xsmadd(a|m)sp,” on page 575.

The intermediate result is rounded to single-precision using the rounding mode specified by RN.

See Table 50, “Scalar Floating-Point Intermediate Result Handling,” on page 515.

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 undefined.

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 51, “VSX Scalar Floating-Point Final Result,” on page 516.

---

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.
Special Registers Altered

<table>
<thead>
<tr>
<th>FPRF</th>
<th>FR</th>
<th>FI</th>
<th>FX</th>
<th>OX</th>
<th>UX</th>
<th>XX</th>
</tr>
</thead>
<tbody>
<tr>
<td>VXSNAN</td>
<td>VXISI</td>
<td>VXIMZ</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

VSR Data Layout for xsmadd(a|m)sp

- src1 = VSR[XA]
  
<table>
<thead>
<tr>
<th>DP</th>
<th>unused</th>
</tr>
</thead>
</table>

- src2 = \texttt{xsmaddsp} ? VSR[XT] : VSR[XB]
  
<table>
<thead>
<tr>
<th>DP</th>
<th>unused</th>
</tr>
</thead>
</table>

- src3 = \texttt{xsmaddsp} ? VSR[XB] : VSR[XT]
  
<table>
<thead>
<tr>
<th>DP</th>
<th>unused</th>
</tr>
</thead>
</table>

- tgt = VSR[XT]
  
<table>
<thead>
<tr>
<th>DP</th>
<th>undefined</th>
</tr>
</thead>
</table>

0  64  127
### Table 70. Actions for xsmadd(a|m)sp

#### 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].
- **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.
- **M(x,y)**: Return the intermediate product having unbounded range and precision.
- **p**: The intermediate result having unbounded range and precision.

#### 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>–NZF</td>
<td>p ← –Infinity</td>
<td>p ← M(src1,src3)</td>
<td>p ← –Zero</td>
<td>p ← M(src1,src3)</td>
<td>p ← –Infinity</td>
<td>p ← src3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+Zero</td>
<td>p ← dQNaN</td>
<td>p ← +Zero</td>
<td>p ← dQNaN</td>
<td>p ← +Zero</td>
<td>p ← dQNaN</td>
<td>p ← src3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+NZF</td>
<td>p ← –Infinity</td>
<td>p ← M(src1,src3)</td>
<td>p ← –Zero</td>
<td>p ← M(src1,src3)</td>
<td>p ← +Infinity</td>
<td>p ← src3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+Infinity</td>
<td>p ← –Infinity</td>
<td>p ← M(src1,src3)</td>
<td>p ← –Zero</td>
<td>p ← M(src1,src3)</td>
<td>p ← +Infinity</td>
<td>p ← src3</td>
<td></td>
<td></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></td>
<td></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></td>
<td></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></td>
<td></td>
</tr>
<tr>
<td>–NZF</td>
<td>v ← –Infinity</td>
<td>v ← A(p,src2)</td>
<td>v ← p</td>
<td>v ← A(p,src2)</td>
<td>v ← +Infinity</td>
<td>v ← p</td>
<td></td>
<td></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></td>
<td></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></td>
<td></td>
</tr>
<tr>
<td>+NZF</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></td>
<td></td>
</tr>
<tr>
<td>+Infinity</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></td>
<td></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></td>
<td></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></td>
<td></td>
</tr>
</tbody>
</table>
**VSX Scalar Multiply-Add Quad-Precision [using round to Odd] X-form**

<table>
<thead>
<tr>
<th>xsmaddqp</th>
<th>VRT, VRA, VRB</th>
</tr>
</thead>
<tbody>
<tr>
<td>xsmaddqpo</td>
<td>VRT, VRA, VRB</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>388</th>
<th>458</th>
</tr>
</thead>
<tbody>
<tr>
<td>RO</td>
<td>63</td>
<td>6</td>
<td>VRT</td>
<td>VRA</td>
<td>VRB</td>
<td>388</td>
<td>458</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, src2)

rd ← bfp_ROUND_TO_BFP128(RO, FPSCR.RN, v)

result ← bfp_CONVERT_TO_BFP128(rd)

if vxsnan_flag then SetFX(FPSCR.VXSNAN)
if vximz_flag then SetFX(FPSCR.VXIMZ)
if vxisi_flag then SetFX(FPSCR.VXISI)
if vx_flag then SetFX(FPSCR.VX)
if vx_flag then SetFX(FPSCR.VX)
if vx_flag then SetFX(FPSCR.VX)

vx_flag ← vxsnan_flag | vximz_flag | vxisi_flag
ex_flag ← FPSCR.EE & vx_flag

if ex_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 VXIZER 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 69. "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 69. "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.

See Table 50, “Scalar Floating-Point Intermediate Result Handling,” on page 515.

The result is placed into VSR[VRT+32] in quad-precision format.

---

1. The quad-precision default Quiet NaN is the value, 0x7FFF_8000_0000_0000_0000_0000_0000.
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 51, “VSX Scalar Floating-Point Final Result,” on page 516.

**Special Registers Altered:**

FPRF, FR, FI
FX VXSNAN VXIMZ VXISI OX UX XX

**VSR Data Layout for xsmaddqp[o]**

<table>
<thead>
<tr>
<th>VSR[VRX+32]</th>
<th>src1</th>
</tr>
</thead>
<tbody>
<tr>
<td>VSR[VRT+32]</td>
<td>src2</td>
</tr>
<tr>
<td>VSR[VRS+32]</td>
<td>src3</td>
</tr>
<tr>
<td>VSR[VRB+32]</td>
<td>tgt</td>
</tr>
</tbody>
</table>
**Part 1: Multiply**

<table>
<thead>
<tr>
<th>src1</th>
<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>(p) &amp; rightarrow; (-\infty)</td>
<td>(p) &amp; rightarrow; QNaN; vximz_flag &amp; rightarrow; 1</td>
<td>(p) &amp; rightarrow; (-\text{Infinity})</td>
<td>(p) &amp; rightarrow; QNaN; vximz_flag &amp; rightarrow; 1</td>
<td>(p) &amp; rightarrow; (-\text{Infinity})</td>
<td>(p) &amp; rightarrow; QNaN; vximz_flag &amp; rightarrow; 1</td>
<td>(p) &amp; rightarrow; (-\text{Infinity})</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(-\text{NZF})</td>
<td>(p) &amp; rightarrow; QNaN; vximz_flag &amp; rightarrow; 1</td>
<td>(p) &amp; rightarrow; (-\text{Infinity})</td>
<td>(p) &amp; rightarrow; QNaN; vximz_flag &amp; rightarrow; 1</td>
<td>(p) &amp; rightarrow; (-\text{Infinity})</td>
<td>(p) &amp; rightarrow; QNaN; vximz_flag &amp; rightarrow; 1</td>
<td>(p) &amp; rightarrow; (-\text{Infinity})</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(-\text{Zero})</td>
<td>(p) &amp; rightarrow; QNaN; vximz_flag &amp; rightarrow; 1</td>
<td>(p) &amp; rightarrow; (-\text{Infinity})</td>
<td>(p) &amp; rightarrow; QNaN; vximz_flag &amp; rightarrow; 1</td>
<td>(p) &amp; rightarrow; (-\text{Infinity})</td>
<td>(p) &amp; rightarrow; QNaN; vximz_flag &amp; rightarrow; 1</td>
<td>(p) &amp; rightarrow; (-\text{Infinity})</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(+\text{Zero})</td>
<td>(p) &amp; rightarrow; QNaN; vximz_flag &amp; rightarrow; 1</td>
<td>(p) &amp; rightarrow; (-\text{Infinity})</td>
<td>(p) &amp; rightarrow; QNaN; vximz_flag &amp; rightarrow; 1</td>
<td>(p) &amp; rightarrow; (-\text{Infinity})</td>
<td>(p) &amp; rightarrow; QNaN; vximz_flag &amp; rightarrow; 1</td>
<td>(p) &amp; rightarrow; (-\text{Infinity})</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(+\text{NZF})</td>
<td>(p) &amp; rightarrow; QNaN; vximz_flag &amp; rightarrow; 1</td>
<td>(p) &amp; rightarrow; (-\text{Infinity})</td>
<td>(p) &amp; rightarrow; QNaN; vximz_flag &amp; rightarrow; 1</td>
<td>(p) &amp; rightarrow; (-\text{Infinity})</td>
<td>(p) &amp; rightarrow; QNaN; vximz_flag &amp; rightarrow; 1</td>
<td>(p) &amp; rightarrow; (-\text{Infinity})</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(+\infty)</td>
<td>(p) &amp; rightarrow; QNaN; vximz_flag &amp; rightarrow; 1</td>
<td>(p) &amp; rightarrow; (-\text{Infinity})</td>
<td>(p) &amp; rightarrow; QNaN; vximz_flag &amp; rightarrow; 1</td>
<td>(p) &amp; rightarrow; (-\text{Infinity})</td>
<td>(p) &amp; rightarrow; QNaN; vximz_flag &amp; rightarrow; 1</td>
<td>(p) &amp; rightarrow; (-\text{Infinity})</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>QNaN</td>
<td>(p) &amp; rightarrow; QNaN; vximz_flag &amp; rightarrow; 1</td>
<td>(p) &amp; rightarrow; (-\text{Infinity})</td>
<td>(p) &amp; rightarrow; QNaN; vximz_flag &amp; rightarrow; 1</td>
<td>(p) &amp; rightarrow; (-\text{Infinity})</td>
<td>(p) &amp; rightarrow; QNaN; vximz_flag &amp; rightarrow; 1</td>
<td>(p) &amp; rightarrow; (-\text{Infinity})</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SNaN</td>
<td>(p) &amp; rightarrow; QNaN; vximz_flag &amp; rightarrow; 1</td>
<td>(p) &amp; rightarrow; (-\text{Infinity})</td>
<td>(p) &amp; rightarrow; QNaN; vximz_flag &amp; rightarrow; 1</td>
<td>(p) &amp; rightarrow; (-\text{Infinity})</td>
<td>(p) &amp; rightarrow; QNaN; vximz_flag &amp; rightarrow; 1</td>
<td>(p) &amp; rightarrow; (-\text{Infinity})</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Part 2: Add**

<table>
<thead>
<tr>
<th>src1</th>
<th>src2</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>(v) &amp; rightarrow; (-\infty)</td>
<td>(v) &amp; rightarrow; QNaN; vxisi_flag &amp; rightarrow; 1</td>
<td>(v) &amp; rightarrow; (-\text{Infinity})</td>
<td>(v) &amp; rightarrow; QNaN; vxisi_flag &amp; rightarrow; 1</td>
<td>(v) &amp; rightarrow; (-\text{Infinity})</td>
<td>(v) &amp; rightarrow; QNaN; vxisi_flag &amp; rightarrow; 1</td>
<td>(v) &amp; rightarrow; (-\text{Infinity})</td>
<td>(v) &amp; rightarrow; (-\text{Infinity})</td>
<td></td>
</tr>
<tr>
<td>(-\text{NZF})</td>
<td>(v) &amp; rightarrow; QNaN; vxisi_flag &amp; rightarrow; 1</td>
<td>(v) &amp; rightarrow; (-\text{Infinity})</td>
<td>(v) &amp; rightarrow; QNaN; vxisi_flag &amp; rightarrow; 1</td>
<td>(v) &amp; rightarrow; (-\text{Infinity})</td>
<td>(v) &amp; rightarrow; QNaN; vxisi_flag &amp; rightarrow; 1</td>
<td>(v) &amp; rightarrow; (-\text{Infinity})</td>
<td>(v) &amp; rightarrow; (-\text{Infinity})</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(-\text{Zero})</td>
<td>(v) &amp; rightarrow; QNaN; vxisi_flag &amp; rightarrow; 1</td>
<td>(v) &amp; rightarrow; (-\text{Infinity})</td>
<td>(v) &amp; rightarrow; QNaN; vxisi_flag &amp; rightarrow; 1</td>
<td>(v) &amp; rightarrow; (-\text{Infinity})</td>
<td>(v) &amp; rightarrow; QNaN; vxisi_flag &amp; rightarrow; 1</td>
<td>(v) &amp; rightarrow; (-\text{Infinity})</td>
<td>(v) &amp; rightarrow; (-\text{Infinity})</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(+\text{Zero})</td>
<td>(v) &amp; rightarrow; QNaN; vxisi_flag &amp; rightarrow; 1</td>
<td>(v) &amp; rightarrow; (-\text{Infinity})</td>
<td>(v) &amp; rightarrow; QNaN; vxisi_flag &amp; rightarrow; 1</td>
<td>(v) &amp; rightarrow; (-\text{Infinity})</td>
<td>(v) &amp; rightarrow; QNaN; vxisi_flag &amp; rightarrow; 1</td>
<td>(v) &amp; rightarrow; (-\text{Infinity})</td>
<td>(v) &amp; rightarrow; (-\text{Infinity})</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(+\text{NZF})</td>
<td>(v) &amp; rightarrow; QNaN; vxisi_flag &amp; rightarrow; 1</td>
<td>(v) &amp; rightarrow; (-\text{Infinity})</td>
<td>(v) &amp; rightarrow; QNaN; vxisi_flag &amp; rightarrow; 1</td>
<td>(v) &amp; rightarrow; (-\text{Infinity})</td>
<td>(v) &amp; rightarrow; QNaN; vxisi_flag &amp; rightarrow; 1</td>
<td>(v) &amp; rightarrow; (-\text{Infinity})</td>
<td>(v) &amp; rightarrow; (-\text{Infinity})</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(+\infty)</td>
<td>(v) &amp; rightarrow; QNaN; vxisi_flag &amp; rightarrow; 1</td>
<td>(v) &amp; rightarrow; (-\text{Infinity})</td>
<td>(v) &amp; rightarrow; QNaN; vxisi_flag &amp; rightarrow; 1</td>
<td>(v) &amp; rightarrow; (-\text{Infinity})</td>
<td>(v) &amp; rightarrow; QNaN; vxisi_flag &amp; rightarrow; 1</td>
<td>(v) &amp; rightarrow; (-\text{Infinity})</td>
<td>(v) &amp; rightarrow; (-\text{Infinity})</td>
<td></td>
<td></td>
</tr>
<tr>
<td>QNaN &amp; src1 is a NaN</td>
<td>(v) &amp; rightarrow; QNaN; vxisi_flag &amp; rightarrow; 1</td>
<td>(v) &amp; rightarrow; (-\text{Infinity})</td>
<td>(v) &amp; rightarrow; QNaN; vxisi_flag &amp; rightarrow; 1</td>
<td>(v) &amp; rightarrow; (-\text{Infinity})</td>
<td>(v) &amp; rightarrow; QNaN; vxisi_flag &amp; rightarrow; 1</td>
<td>(v) &amp; rightarrow; (-\text{Infinity})</td>
<td>(v) &amp; rightarrow; (-\text{Infinity})</td>
<td></td>
<td></td>
</tr>
<tr>
<td>QNaN &amp; src1 not a NaN</td>
<td>(v) &amp; rightarrow; QNaN; vxisi_flag &amp; rightarrow; 1</td>
<td>(v) &amp; rightarrow; (-\text{Infinity})</td>
<td>(v) &amp; rightarrow; QNaN; vxisi_flag &amp; rightarrow; 1</td>
<td>(v) &amp; rightarrow; (-\text{Infinity})</td>
<td>(v) &amp; rightarrow; QNaN; vxisi_flag &amp; rightarrow; 1</td>
<td>(v) &amp; rightarrow; (-\text{Infinity})</td>
<td>(v) &amp; rightarrow; (-\text{Infinity})</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[VRT+32].
- **src3:** The quad-precision floating-point value in VSR[VRA+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.
- **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.
VSX Scalar Maximum Double-Precision XX3-form

\textbf{xsmaxdp} \quad \text{XT,XA,XB}

\begin{center}
\begin{tabular}{|c|c|c|c|c|c|}
\hline
 & 60 & 6 & 11 & 16 & 21 & 160 \\
\hline
XT & $\leftarrow TX || T$ & & & & & \\
XA & $\leftarrow AX || A$ & & & & & \\
XB & $\leftarrow BX || B$ & & & & & \\
\hline
\end{tabular}
\end{center}

reset_xflags();

\begin{itemize}
\item \texttt{src1} $\leftarrow \text{VSR}[XA][0:63]$
\item \texttt{src2} $\leftarrow \text{VSR}[XB][0:63]$
\item \texttt{result}[0:63] $\leftarrow \text{MaximumDP}($\texttt{src1},\texttt{src2}$)$
\item if (vxsnan_flag) then SetFX(VXSNAN)
\end{itemize}

\begin{itemize}
\item vex_flag $\leftarrow \text{VE} \& \text{vxsnan_flag}$
\end{itemize}

if (~vex_flag) then do

\begin{itemize}
\item \texttt{VSR}[XT] $\leftarrow \text{result} || 0xUUUU_UUUU_UUUU_UUUU$
\end{itemize}

end

Let \texttt{XT} be the value $32 \times TX + T$.
Let \texttt{XA} be the value $32 \times AX + A$.
Let \texttt{XB} be the value $32 \times BX + B$.

Let \texttt{src1} be the double-precision floating-point value in doubleword element 0 of VSR[XA].

Let \texttt{src2} be the double-precision floating-point value in doubleword element 0 of VSR[XB].

If \texttt{src1} is greater than \texttt{src2}, \texttt{src1} is placed into doubleword element 0 of VSR[XT]. Otherwise, \texttt{src2} is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are undefined.

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.

FPRF, FR and FI are not modified.

If a trap-enabled invalid operation exception occurs, VSR[XT] is not modified.

See Table 72.

Special Registers Altered

\begin{itemize}
\item FX \quad VXSNAN
\end{itemize}

\begin{itemize}
\item Programming Note
This instruction can be used to operate on single-precision source operands.
\end{itemize}
<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(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(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(src1)</td>
<td>T(src1)</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(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(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(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>
</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.
- **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 72. Actions for xsmxdp**
**VSX Scalar Maximum Type-C Double-Precision XX3-form**

$$\text{xsmaxcdp } \ XT, \XA, \XB$$

<table>
<thead>
<tr>
<th>60</th>
<th>T</th>
<th>A</th>
<th>B</th>
<th>128</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>11</td>
<td>10</td>
<td>p1</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 $VSR[AX]$.
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 undefined.

If a trap-enabled Invalid Operation occurs, $VSR[XT]$ is not modified.

**Special Registers Altered:**

- FX VXSNAN
### Table 73: Actions for `xsmacdp`

<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(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(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(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(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(src1)</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 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 undefined.
- `FXR, FR, and FI` are not modified.
- `fx(x)` If `x` is equal to 0, `FX` is set to 1; otherwise, `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 Maximum Type-J 
Double-Precision XX3-form 

Let $X_T$ be the value $32 \times TX + T$.
Let $X_A$ be the value $32 \times AX + A$.
Let $X_B$ be the value $32 \times BX + B$.

Let $src_1$ be the double-precision floating-point value in doubleword 0 of $VSR[XA]$.
Let $src_2$ be the double-precision floating-point value in doubleword 0 of $VSR[XB]$.

If $src_1$ or $src_2$ is a SNaN, an Invalid Operation exception occurs.
If $src_1$ is a NaN, result is $src_1$.
Otherwise, if $src_2$ is a NaN, result is $src_2$.
Otherwise, if $src_1$ is a Zero and $src_2$ is a Zero and either $src_1$ or $src_2$ is a +Zero, the result is +Zero.
Otherwise, if $src_1$ is a -Zero and $src_2$ is a -Zero, the result is -Zero.
Otherwise, if $src_1$ is greater than $src_2$, result is $src_1$.
Otherwise, result is $src_2$.

The contents of doubleword 0 of $VSR[XT]$ are set to the value result.
The contents of doubleword 1 of $VSR[XT]$ are undefined.

If a trap-enabled Invalid Operation occurs, $VSR[XT]$ is not modified.

Special Registers Altered:
FX VXSNAN
### Table 74. Actions for xsmajdp

<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>
<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>
<th>src2</th>
<th>src1</th>
<th>src2</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(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>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>−NZF</td>
<td>T(src1)</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>
<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>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(+Zero)</td>
<td>T(+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>
<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>T(src1)</td>
<td>T(src1)</td>
</tr>
<tr>
<td>+Zero</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(+Zero)</td>
<td>T(+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>
<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>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(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>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>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(+INF)</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>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(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>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>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>
<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>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.
- **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. If `VE=1`, update of `VSR[XT]` is suppressed.
VSX Scalar Minimum Double-Precision
XX3-form

xsmindp XT,XA,XB

<table>
<thead>
<tr>
<th>60</th>
<th>T</th>
<th>A</th>
<th>B</th>
<th>168</th>
<th>179</th>
</tr>
</thead>
</table>

XT ← TX || T
XA ← AX || A
XB ← BX || B
reset_xflags() src1 ← VSR[XA](0:63)
src2 ← VSR[XB](0:63)
result[0:63] ← MinimumDP(src1,src2)
if(vxsnan_flag) then SetFX(VXSNAN)
ve_flag ← VE & vxsnan_flag
if( ~ve_flag ) then do
    VSR[XT] ← result || 0xUUUU_UUUU_UUUU_UUUU
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 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 undefined.

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 75.

Special Registers Altered

FX VXSNAN

Programming Note

This instruction can be used to operate on single-precision source operands.
<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>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 undefined. 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 75. Actions for xvmindp**
**VSX Scalar Minimum Type-C Double-Precision XX3-form**

\[ \text{xmincdp} \quad \text{XT,XA,XB} \]

- **T**
- **A**
- **B**
- **result**
- **src1**
- **src2**
- **VF**
- **FX**
- **VXSNAN**

<table>
<thead>
<tr>
<th>( 60 )</th>
<th>( 11 )</th>
<th>( 10 )</th>
<th>( 9 )</th>
<th>( 136 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>A</td>
<td>B</td>
<td>src1</td>
<td>src2</td>
</tr>
</tbody>
</table>

\[
\text{Let } \text{XT} \text{ be the value } 32 \times \text{TX} + T. \\
\text{Let } \text{XA} \text{ be the value } 32 \times \text{AX} + A. \\
\text{Let } \text{XB} \text{ be the value } 32 \times \text{BX} + B. \\
\]

\[
\text{Let } \text{src1} \text{ be the double-precision floating-point value in doubleword 0 of VSR[} \text{XA}]. \\
\text{Let } \text{src2} \text{ be the double-precision floating-point value in doubleword 0 of VSR[} \text{XB}]. \\
\]

\[
\text{If } \text{src1} \text{ or } \text{src2} \text{ is a SNaN, an Invalid Operation exception occurs.} \\
\text{If either } \text{src1} \text{ or } \text{src2} \text{ is a NaN, result is } \text{src2}. \\
\text{Otherwise, if } \text{src1} \text{ is less than } \text{src2}, \text{result is } \text{src1}. \\
\text{Otherwise, result is } \text{src2}. \\
\]

\[
\text{The contents of doubleword 0 of VSR[} \text{XT}] \text{ are set to the value result.} \\
\text{The contents of doubleword 1 of VSR[} \text{XT}] \text{ are undefined.} \\
\text{If a trap-enabled Invalid Operation occurs, VSR[} \text{XT}] \text{ is not modified.} \\
\]

**Special Registers Altered:**

\[
\text{FX VXSNAN} \\
\]
**Table 76. 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(src1)</td>
<td>T(src1)</td>
<td>T(src1)</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(src1)</td>
<td>T(src1)</td>
<td>T(src1)</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(src1)</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(M(src1, 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 undefined.
- **FX**: Set to 1 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.

\( VSR[\text{XX}] \) elements have undefined values for this instruction.
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.
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 undefined.

If a trap-enabled Invalid Operation occurs, VSR[XT] is not modified.

Special Registers Altered:
FX VXSNAN
Table 77. 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(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(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src2)</td>
<td>T(src1)</td>
<td>T(src1)</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(src2)</td>
<td>T(src1)</td>
<td>T(src1)</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(src2)</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(M(src1,src2))</td>
<td>T(src1)</td>
<td>T(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(+INF)</td>
<td>T(src2)</td>
<td>T(src1)</td>
<td>T(src1)</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>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>
<td>T(src1)</td>
</tr>
<tr>
<td></td>
<td>fx(VXSNAN)</td>
<td>fx(VXSNAN)</td>
<td>fx(VXSNAN)</td>
<td>fx(VXSNAN)</td>
<td>fx(VXSNAN)</td>
<td>fx(VXSNAN)</td>
<td>fx(VXSNAN)</td>
<td>fx(VXSNAN)</td>
<td>fx(VXSNAN)</td>
</tr>
</tbody>
</table>

Explanation:

- **src1**: The double-precision floating-point value in doubleword element 0 of VSR[EA].
- **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.
- **VXSNAN**: Floating-Point Invalid Operation Exception (SNaN) status flag, VXSNAN. If VE = 1, update of VSR[XT] is suppressed.
For \textit{xsmsubadp}, do the following.

- Let \textit{src1} be the double-precision floating-point value in doubleword element 0 of \text{VSR[XA]}.
- Let \textit{src2} be the double-precision floating-point value in doubleword element 0 of \text{VSR[XB]}.
- Let \textit{src3} be the double-precision floating-point value in doubleword element 0 of \text{VSR[XT]}.

\textit{src1} is multiplied\textsuperscript{[1]} by \textit{src3}, producing a product having unbounded range and precision.

See part 1 of Table 78.

\textit{src2} is negated and added\textsuperscript{[2]} to the product, producing a sum having unbounded range and precision.

The result, having unbounded range and precision, is normalized\textsuperscript{[3]}.

See part 2 of Table 78.

The intermediate result is rounded to double-precision using the rounding mode specified by \text{RN}.

See Table 50, “Scalar Floating-Point Intermediate Result Handling,” on page 515.

The result is placed into doubleword element 0 of \text{VSR[XT]} in double-precision format.

The contents of doubleword element 1 of \text{VSR[XT]} are undefined.

\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.

If a trap-enabled invalid operation exception occurs, \text{VSR[XT]} and \text{FPRF} are not modified, and \text{FR} and \text{FI} are set to 0.

See Table 51, “VSX Scalar Floating-Point Final Result,” on page 516.

\textbf{Special Registers Altered}

\text{FPRF FR FX OX UX XX VXSNAN VXISI VXIMZ}

\begin{itemize}
  \item \text{VXSNAN VXISI VXIMZ}
\end{itemize}

\textbf{1.} Floating-point multiplication is based on exponent addition and multiplication of the significands.

\textbf{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.

\textbf{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 `xsmsub(a|m)dp`

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>src1</td>
<td>VSR[XA]</td>
<td>DP</td>
</tr>
<tr>
<td>tgt</td>
<td>VSR[XT]</td>
<td>DP</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>64</th>
<th>127</th>
</tr>
</thead>
<tbody>
<tr>
<td>DP</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 78. Actions for xsmsub(a|m)dp

<table>
<thead>
<tr>
<th>Part 1: Multiply</th>
<th>src3</th>
<th>+Infinity</th>
<th>+NZF</th>
<th>+Zero</th>
<th>+NZF</th>
<th>+Infinity</th>
<th>QNaN</th>
<th>NaN</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>-Infinity</strong></td>
<td>p ← -Infinity</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>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>-NZF</strong></td>
<td>p ← +Infinity</td>
<td>p ← M(src1,src3)</td>
<td>p ← +Zero</td>
<td>p ← -Infinity</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><strong>-Zero</strong></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 ← src3</td>
</tr>
<tr>
<td><strong>+Zero</strong></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 ← src3</td>
</tr>
<tr>
<td><strong>+NZF</strong></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 ← src3</td>
</tr>
<tr>
<td><strong>+Infinity</strong></td>
<td>p ← -Infinity</td>
<td>p ← +Infinity</td>
<td>p ← -Infinity</td>
<td>p ← -Infinity</td>
<td>p ← src3</td>
<td>p ← src3</td>
<td>p ← src3</td>
<td>p ← src3</td>
</tr>
<tr>
<td><strong>QNaN</strong></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><strong>SNaN</strong></td>
<td>p ← Q(src1) vximz_flag ← 1</td>
<td>p ← Q(src1) vximz_flag ← 1</td>
<td>p ← Q(src1) vximz_flag ← 1</td>
<td>p ← Q(src1) vximz_flag ← 1</td>
<td>p ← Q(src1) vximz_flag ← 1</td>
<td>p ← Q(src1) vximz_flag ← 1</td>
<td>p ← Q(src1) vximz_flag ← 1</td>
<td>p ← src1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Part 2: Subtract</th>
<th>src2</th>
<th>src3</th>
<th>+Infinity</th>
<th>+NZF</th>
<th>+Zero</th>
<th>+NZF</th>
<th>+Infinity</th>
<th>QNaN</th>
<th>NaN</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>-Infinity</strong></td>
<td>v ← dQNaN vximz_flag ← 1</td>
<td>v ← dQNaN vximz_flag ← 1</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← src2</td>
<td>v ← src2</td>
<td>v ← src2</td>
<td></td>
</tr>
<tr>
<td><strong>-NZF</strong></td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← src2</td>
<td>v ← src2</td>
<td>v ← src2</td>
<td></td>
</tr>
<tr>
<td><strong>-Zero</strong></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 ← src2</td>
<td></td>
</tr>
<tr>
<td><strong>+Zero</strong></td>
<td>v ← +Infinity</td>
<td>v ← -Infinity</td>
<td>v ← vxisi_flag</td>
<td>v ← vxisi_flag</td>
<td>v ← vxisi_flag</td>
<td>v ← src2</td>
<td>v ← src2</td>
<td>v ← src2</td>
<td></td>
</tr>
<tr>
<td><strong>+NZF</strong></td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← src2</td>
<td>v ← src2</td>
<td>v ← src2</td>
<td></td>
</tr>
<tr>
<td><strong>+Infinity</strong></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 ← src2</td>
<td></td>
</tr>
<tr>
<td><strong>QNaN &amp; src1 is a NaN</strong></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 ← src2</td>
<td>v ← src2</td>
<td></td>
</tr>
<tr>
<td><strong>QNaN &amp; src1 not a NaN</strong></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 ← src2</td>
<td>v ← src2</td>
<td></td>
</tr>
</tbody>
</table>

### Explanation:

- **src1**: The double-precision floating-point value in doubleword element 0 of VSR[XA].
- **src2**: For **xsmsubdp**, the double-precision floating-point value in doubleword element 0 of VSR[XT].
- **src3**: For **xsmsubdp**, 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.
- **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 Single-Precision XX3-form**

**xsmsubasp** XT,XA,XB

<table>
<thead>
<tr>
<th>60</th>
<th>T</th>
<th>A</th>
<th>B</th>
<th>17</th>
<th>k kB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>21</td>
</tr>
</tbody>
</table>

**xsmsubmsp** XT,XA,XB

| 60 |  T |  A |  B | 25 | k kB| |
|----|----|----|----|----|----|
| 0  | 6  | 11 | 16 | 21 | 21 |

```plaintext
reset_xflags()

if "xsmsubasp" then do
    src1 ← VSR[32×AX+A].dword[0]
    src2 ← VSR[32×TX+T].dword[0]
    src3 ← VSR[32×BX+B].dword[0]
end

if "xsmsubmsp" then do
    src1 ← VSR[32×AX+A].dword[0]
    src2 ← VSR[32×BX+B].dword[0]
    src3 ← VSR[32×TX+T].dword[0]
end

v ← MultiplyAddDP(src1,src3,NegateDP(src2))
result ← RoundToSP(RN,v)
if(vxsnan_flag) then SetFX(VXSNAN)
if(vximz_flag)  then SetFX(VXIMZ)
if(vxisi_flag)  then SetFX(VXISI)
if(ox_flag)     then SetFX(OX)
if(ux_flag)     then SetFX(UX)
if(xx_flag)     then SetFX(XX)
vex_flag ← VE & (vxsnan_flag | vximz_flag | vxisi_flag)
if( ~vex_flag ) then do
    VSR[32×TX+T].dword[0] ← ConvertSPtoSP64(result)
    VSR[32×TX+T].dword[1] ← 0xUUUU_UUUU_UUUU_UUUU
    FPRF ← ClassSP(result)
    FR ← inc_flag
    FI ← xx_flag
end else do
    FR ← 0b0
    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.
```

For **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[XB].

For **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[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.
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 1 of Table 79, “Actions for xsmsub(a|m)sp”.

The intermediate result is rounded to single-precision using the rounding mode specified by RN.

See Table 50, “Scalar Floating-Point Intermediate Result Handling,” on page 515.

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 undefined.

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 51, “VSX Scalar Floating-Point Final Result,” on page 516.

---

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.
Special Registers Altered

FPRF  FR  FI  FX  OX  UX  XX
VXSNAN  VXISI  VXIMZ

VSR Data Layout for xsmsub(a|m)sp

src1 = VSR[XA]

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>DP</td>
<td></td>
<td>unused</td>
</tr>
</tbody>
</table>

src2 = \textit{xsmsubasp} ? VSR[XT] : VSR[XB]

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>DP</td>
<td></td>
<td>unused</td>
</tr>
</tbody>
</table>

src3 = \textit{xsmsubasp} ? VSR[XB] : VSR[XT]

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>DP</td>
<td></td>
<td>unused</td>
</tr>
</tbody>
</table>

tgt = VSR[XT]

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>DP</td>
<td></td>
<td>undefined</td>
</tr>
</tbody>
</table>

0 64 127
### Part 1: Multiply

<table>
<thead>
<tr>
<th>src3</th>
<th>–Infinity</th>
<th>–NZF</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) 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 ← Q(src3) vxsnan_flag ← 1</td>
</tr>
<tr>
<td>+Zero</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 ← 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 ← Q(src3) vxsnan_flag ← 1</td>
</tr>
<tr>
<td>+Infinity</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 ← 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 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>
</tr>
</tbody>
</table>

### Part 2: Subtract

<table>
<thead>
<tr>
<th>src2</th>
<th>–Infinity</th>
<th>–NZF</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 ← –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>
</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>
</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>
</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>
</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>
</tr>
<tr>
<td>QNaN &amp; src1 is a NaN</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 not a NaN</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>
</tbody>
</table>

**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].
- src3: For `xsmsubasp`, 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.
- The intermediate product having unbounded range and precision.
- The intermediate result having unbounded range and precision.

Table 79. Actions for `xsmsub(a|m)`sp
### VSX Scalar Multiply-Subtract Quad-Precision [using round to Odd] X-form

<table>
<thead>
<tr>
<th>xmsubqp</th>
<th>VRT, VRA, VRB</th>
<th>(RO=0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmsubqpo</td>
<td>VRT, VRA, VRB</td>
<td>(RO=1)</td>
</tr>
</tbody>
</table>

#### Algorithm

1. **Let** src1 be the floating-point value in VSR[VRA+32] represented in quad-precision format.
2. **Let** src2 be the floating-point value in VSR[VRT+32] represented in quad-precision format.
3. **Let** src3 be the floating-point value in VSR[VRB+32] represented in quad-precision format.
4. If either src1, src2, or src3 is a Signalling NaN, an Invalid Operation exception occurs and VXSNAN is set to 1.
5. 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 the Quiet NaN corresponding to src3.
6. 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.
7. If src1 is a Signalling NaN, the result is the Quiet NaN corresponding to src1.
8. Otherwise, if src1 is a Quiet NaN, the result is src1.
9. Otherwise, if src2 is a Signalling NaN, the result is the Quiet NaN corresponding to src2.
10. Otherwise, if src2 is a Quiet NaN, the result is src2.
11. Otherwise, if src3 is a Signalling NaN, the result is the Quiet NaN corresponding to src3.
12. Otherwise, if src3 is a Quiet NaN, the result is src3.
13. 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.
14. Otherwise, if the product of src1 and src3, and src2 are Infinity values having same signs, the result is the default Quiet NaN.
15. Otherwise, do the following.
   - src1 is multiplied by src3, producing a product having unbounded significand precision and exponent range.
   - See part 1 of Table 80. "Actions for xmsubqp[o]."
   - src2 is negated and added to the product, producing a sum having unbounded range and precision.
   - See part 2 of Table 80. "Actions for xmsubqp[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.
   - See Table 50, "Scalar Floating-Point Intermediate Result Handling," on page 515.
   - 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.

---

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 51, “VSX Scalar Floating-Point Final Result,” on page 516.

**Special Registers Altered:**

| FPRF | FR  | FI  | FX  | VXSNAN | VXIMZ | VXISI | OX  | UX  | XX  |

**VSR Data Layout for xsmsubqp[o]**

<table>
<thead>
<tr>
<th>VSR(VRA+32)</th>
<th>src1</th>
</tr>
</thead>
<tbody>
<tr>
<td>VSR(VRT+32)</td>
<td>src2</td>
</tr>
<tr>
<td>VSR(VRB+32)</td>
<td>src3</td>
</tr>
<tr>
<td>VSR(VRT+32)</td>
<td>tgt</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></td>
<td>p – QNaN</td>
<td></td>
<td></td>
<td></td>
<td>p – infinity</td>
<td></td>
</tr>
<tr>
<td>–NZF</td>
<td></td>
<td>p + infinite</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>–Zero</td>
<td>p – QNaN</td>
<td></td>
<td>p + zero</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>p + QNaN</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>p + src1</td>
<td></td>
</tr>
<tr>
<td>SNaN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>p + quiet(src1)</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 &amp; src1 is a NaN</th>
<th>QNaN &amp; src1 not a NaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>–Infinity</td>
<td>v – dQNaN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v – infinity</td>
<td></td>
<td></td>
</tr>
<tr>
<td>–NZF</td>
<td></td>
<td>v – sub(p, src2)</td>
<td></td>
<td>v – p</td>
<td>v – sub(p, src2)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+Zero</td>
<td>v – sub(p, src2)</td>
<td>v – p</td>
<td>v – sub(p, src2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+NZF</td>
<td>v – infinity</td>
<td></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 – p</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v – src2</td>
</tr>
<tr>
<td>QNaN &amp; src1 not a NaN</td>
<td>v – src2</td>
<td>v – quiet(src2)</td>
<td></td>
<td>v – src2</td>
<td>v – quiet(src2)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Explanation:

- **src1**: The quad-precision floating-point value in VSR[VR[A+32]].
- **src2**: The quad-precision floating-point value in VSR[VR[B+32]].
- **src3**: The quad-precision floating-point value in VSR[VR[B+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 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 80. Actions for xsmsubqp[o] |
VSX Scalar Multiply Double-Precision

**XX3-form**

```plaintext
xsmuldp  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 $32 \times BX + B$.

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 by $src2$, producing a product having unbounded range and precision.

The product is normalized.

See Table 81.

The intermediate result is rounded to double-precision using the rounding mode specified by $RN$.

See Table 50, “Scalar Floating-Point Intermediate Result Handling,” on page 515.

The result is placed into doubleword element 0 of VSR[XT] in double-precision format.

---

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 81. Actions for xsmuldp

<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 ← +Zero</td>
<td>v ← Q(src1)</td>
<td>v ← +Zero</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 ← +Zero</td>
<td>v ← Q(src1)</td>
<td>v ← +Zero</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 ← +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>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 Multiply Quad-Precision [using round to Odd] X-form**

\[
xsmulpq \quad \text{VRT, VRA, VRB} \quad \text{(RO=0)}
\]

\[
xsmulqpo \quad \text{VRT, VRA, VRB} \quad \text{(RO=1)}
\]

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>36</th>
<th>41</th>
</tr>
</thead>
<tbody>
<tr>
<td>VRT</td>
<td>63</td>
<td>VRA</td>
<td>VRB</td>
<td>36</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Let \( \text{src1} \) be the floating-point value in \( \text{VSR}[\text{VRA}+32] \) represented in quad-precision format.

Let \( \text{src2} \) be the floating-point value in \( \text{VSR}[\text{VRB}+32] \) represented in quad-precision format.

If either \( \text{src1} \) or \( \text{src2} \) is a Signalling NaN, an Invalid Operation exception occurs and \( \text{VXSNAN} \) is set to 1.

If \( \text{src1} \) is an Infinity value and \( \text{src2} \) is a Zero value, or if \( \text{src1} \) is a Zero value and \( \text{src2} \) is an Infinity value, the result is the default Quiet NaN[1].

Otherwise, do the following.

The normalized product of \( \text{src1} \) multiplied by \( \text{src2} \) is produced with unbounded significand precision and exponent range.

See Table 82, "Actions for xsmulpq[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 \( \text{RO}=1 \), let the rounding mode be Round to Odd. Otherwise, let the rounding mode be specified by \( \text{RN} \). Unless the result is an Infinity or a Zero, the intermediate result is rounded to quad-precision using the specified rounding mode.

See Table 50, "Scalar Floating-Point Intermediate Result Handling," on page 515.

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, \( \text{FR} \) 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 51, "VSX Scalar Floating-Point Final Result," on page 516.

**Special Registers Altered:**

\[
\text{FPRF} \quad \text{FR} \quad \text{FI} \quad \text{FX} \quad \text{VXSNAN} \quad \text{VXIMZ} \quad \text{OX} \quad \text{UX} \quad \text{XX}
\]

---

1. The quad-precision default Quiet NaN is the value, \( 0xFFF8_0000_0000_0000_0000_0000_0000 \).
### VSR Data Layout for xsmulqp[o]

<table>
<thead>
<tr>
<th>VSR[VRA+32]</th>
<th>src1</th>
</tr>
</thead>
<tbody>
<tr>
<td>VSR[VRB+32]</td>
<td>src2</td>
</tr>
<tr>
<td>VSR[VRT+32]</td>
<td>tgt</td>
</tr>
</tbody>
</table>

#### Table 82. Actions for xsmulqp[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 ← -Infinity</td>
<td>v ← dQNaN</td>
<td>v ← -Infinity</td>
<td>v ← dQNaN</td>
<td>v ← +Infinity</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-NZF</td>
<td>v ← mul(src1, src2)</td>
<td>v ← -Zero</td>
<td>v ← -Zero</td>
<td>v ← +Infinity</td>
<td>vximz_flag ← 1</td>
<td></td>
<td></td>
<td></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></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+Zero</td>
<td>v ← dQNaN</td>
<td>v ← +Zero</td>
<td>v ← -Zero</td>
<td>v ← +Infinity</td>
<td>vximz_flag ← 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+NZF</td>
<td>v ← mul(src1, src2)</td>
<td>v ← dQNaN</td>
<td>v ← +Zero</td>
<td>v ← +Zero</td>
<td>v ← +Infinity</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+Infinity</td>
<td>v ← +Infinity</td>
<td>v ← dQNaN</td>
<td>v ← +Zero</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>QNaN</td>
<td>v ← src1</td>
<td>v ← dQNaN</td>
<td>vximz_flag ← 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SNaN</td>
<td>v ← quiet(src1)</td>
<td>vximz_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\[VRB+32\].
- **dQNaN**: Default quiet NaN (0x7FFF_8000_0000_0000_0000_0000_0000).
- **NZF**: Nonzero finite number.
- **mul(x, y)**: The floating-point value \(x\) is multiplied\(^1\) 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

xsmulsp XT,XA,XB

reset_xflags()

src1 ← VSR[32×AX+A].dword[0]
src2 ← VSR[32×BX+B].dword[0]

v ← MultiplyDP(src1,src2)
result ← RoundToSP(RN,v)

if(vxsnan_flag) then SetFX(VXSNAN)
if(vximz_flag)  then SetFX(VXIMZ)
if(ox_flag)     then SetFX(OX)
if(ux_flag)     then SetFX(UX)
if(xx_flag)     then SetFX(XX)

vex_flag ← VE & (vxsnan_flag | vximz_flag)

if( ~vex_flag ) then do
  VSR[32×TX+T].dword[0] ← ConvertSPtoSP64(result)
  VSR[32×TX+T].dword[1] ← 0xUUUU_UUUU_UUUU_UUUU
  FPRF ← ClassSP(result)
  FR ← inc_flag
  FI ← xx_flag
else do
  FR ← 0b0
  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 83, “Actions for xsmulsp,” on page 605.

The intermediate result is rounded to single-precision
using the rounding mode specified by RN.

See Table 50, “Scalar Floating-Point Intermediate
Result Handling,” on page 515.

---

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 83. Actions for `xsmulsp`

<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 ← dQNaN.vximz_flag ← 1</td>
<td>v ← dQNaN.vximz_flag ← 1</td>
<td>v ← −Infinity</td>
<td>v ← −Infinity</td>
<td>v ← src2</td>
<td>v ← Q(src2) vximption_flag ← 1</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 ← −Zero</td>
<td>v ← src2</td>
<td>v ← Q(src2) vximption_flag ← 1</td>
</tr>
<tr>
<td>-Zero</td>
<td>v ← dQNaN.vximz_flag ← 1</td>
<td>v ← +Zero</td>
<td>v ← −Zero</td>
<td>v ← +Zero</td>
<td>v ← src2</td>
<td>v ← src2</td>
<td>v ← src2</td>
<td>v ← src2</td>
</tr>
<tr>
<td>+Zero</td>
<td>v ← dQNaN.vximz_flag ← 1</td>
<td>v ← −Zero</td>
<td>v ← +Zero</td>
<td>v ← +Zero</td>
<td>v ← src2</td>
<td>v ← src2</td>
<td>v ← src2</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 ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← src2</td>
<td>v ← Q(src2) vximption_flag ← 1</td>
</tr>
<tr>
<td>+Infinity</td>
<td>v ← −Infinity</td>
<td>v ← +Infinity</td>
<td>v ← dQNaN.vximz_flag ← 1</td>
<td>v ← dQNaN.vximz_flag ← 1</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← src2</td>
<td>v ← Q(src2) vximption_flag ← 1</td>
</tr>
<tr>
<td>QNaN</td>
<td>v ← Q(src1) vximption_flag ← 1</td>
<td>v ← Q(src1) vximption_flag ← 1</td>
<td>v ← Q(src1) vximption_flag ← 1</td>
<td>v ← Q(src1) vximption_flag ← 1</td>
<td>v ← Q(src1) vximption_flag ← 1</td>
<td>v ← Q(src1) vximption_flag ← 1</td>
<td>v ← Q(src1) vximption_flag ← 1</td>
<td>v ← Q(src1) vximption_flag ← 1</td>
</tr>
<tr>
<td>SNaN</td>
<td>v ← Q(src1) vximption_flag ← 1</td>
<td>v ← Q(src1) vximption_flag ← 1</td>
<td>v ← Q(src1) vximption_flag ← 1</td>
<td>v ← Q(src1) vximption_flag ← 1</td>
<td>v ← Q(src1) vximption_flag ← 1</td>
<td>v ← Q(src1) vximption_flag ← 1</td>
<td>v ← Q(src1) vximption_flag ← 1</td>
<td>v ← Q(src1) vximption_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.
VSX Scalar Negative Absolute Double-Precision XX2-form

\[ \text{xsnabsdp} \quad \text{XT},\text{XB} \]

<table>
<thead>
<tr>
<th>60</th>
<th>6</th>
<th>T</th>
<th>(|)</th>
<th>B</th>
<th>361</th>
<th>B|X</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[\begin{align*}
\text{XT} & \leftarrow 32 \times \text{TX} + \text{T} \\
\text{XB} & \leftarrow 32 \times \text{BX} + \text{B}
\end{align*}\]

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 doubleword element 0 of VSR[\(\text{XB}\)], with bit 0 set to 1, is placed into doubleword element 0 of VSR[\(\text{XT}\)].

The contents of doubleword element 1 of VSR[\(\text{XT}\)] are undefined.

Special Registers Altered:
None

VSR Data Layout for xsnabsdp

\[\begin{align*}
\text{src} & \leftarrow \text{VSR[\(\text{XB}\)]}
\end{align*}\]

\[\begin{array}{c|c}
\text{DP} & \text{unused} \\
\hline
0 & 64 & 127
\end{array}\]

\[\begin{align*}
\text{tgt} & \leftarrow \text{VSR[\(\text{XT}\)]}
\end{align*}\]

\[\begin{array}{c|c}
\text{DP} & \text{undefined} \\
\hline
0 & 64 & 127
\end{array}\]

Programming Note:
This instruction can be used to operate on a single-precision source operand.

VSX Scalar Negative Absolute Quad-Precision X-form

\[ \text{xsnabsqp} \quad \text{VRT},\text{VRB} \]

<table>
<thead>
<tr>
<th>63</th>
<th>VRT</th>
<th>8</th>
<th>VRB</th>
<th>804</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[\begin{align*}
\text{if } \text{MSR.VSX}=0 \text{ then } & \text{VSX.Unavailable()} \\
\text{VSR[VRT+32]} & \leftarrow \text{VSR[VRB+32]} \| 0x8000_0000_0000_0000_0000_0000_0000_0000
\end{align*}\]

Let \(\text{src}\) be the floating-point value in VSR[\(\text{VRB+32}\)] represented in quad-precision format.

The negative absolute value of \(\text{src}\) is placed into VSR[\(\text{VRT+32}\)] in quad-precision format.

Special Registers Altered:
None

VSR Data Layout for xsnabsqp

\[\begin{align*}
\text{VSR[VRB+32]} & \leftarrow \text{src} \\
\text{VSR[VRT+32]} & \leftarrow \text{tgt}
\end{align*}\]
VSX Scalar Negate Double-Precision XX2-form

\[ \text{xsnegdp } XT, XB \]

| 60 | 8 | T | 11 || B | 377 | P[X] |
|----|---|---|---||---|---|----|

\[ XT \leftarrow TX \| T \]
\[ XB \leftarrow BX \| B \]
\[ \text{result}[0:63] \leftarrow \text{VSR}[XB][0] \| \text{VSR}[XB][1:63] \]
\[ \text{VSR}[XT] \leftarrow \text{result} \| 0xUUUU_UUUU_UUUU_UUUU \]

Let \( XT \) be the value \( 32 \times TX + T \).
Let \( XB \) be the value \( 32 \times BX + B \).

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 undefined.

**Special Registers Altered**
None

**VSR Data Layout for xsnegdp**

\[ \text{src} = \text{VSR}[XB] \]
\[ \text{tgt} = \text{VSR}[XT] \]

**Programming Note**
This instruction can be used to operate on a single-precision source operand.

VSX Scalar Negate Quad-Precision X-form

\[ \text{xsnegqp } VRT, VRB \]

<table>
<thead>
<tr>
<th>63</th>
<th>VRT</th>
<th>16</th>
<th>VRB</th>
<th>804</th>
<th>_</th>
</tr>
</thead>
</table>

\[ \text{if MSR.VSX} = 0 \text{ then VSX.Unavailable}() \]
\[ \text{VSR}[VRT+32] \leftarrow \text{VSR}[VRB+32] \| 0x8000_0000_0000_0000_0000_0000_0000_0000 \]

Let \( \text{src} \) be the floating-point value in VSR[VRB+32] represented in quad-precision format.
\( \text{src} \) is negated and placed into VSR[VRT+32] in quad-precision format.

**Special Registers Altered**
None

**VSR Data Layout for xsnegqp**

\[ \text{VSR}[VRB+32] \]
\[ \text{src} \]

\[ \text{VSR}[VRT+32] \]
\[ \text{tgt} \]
VSX Scalar Negative Multiply-Add Double-Precision XX3-form

\[\text{xsnmaddadp} \ XT,XA,XB\]

\[
\begin{array}{cccccc}
\text{T} & \text{A} & \text{B} & 161 & \text{VS} & \text{FR} \\
0 & 6 & 11 & 16 & 21 & \\
\hline
\end{array}
\]

\[\text{xsnmaddmdp} \ XT,XA,XB\]

\[
\begin{array}{cccccc}
\text{T} & \text{A} & \text{B} & 169 & \text{VS} & \text{FR} \\
0 & 6 & 11 & 16 & 21 & \\
\hline
\end{array}
\]

Let \(\text{XT}\) be the value \(32 \times TX + T\).
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[XT].

For \(\text{xsnmaddadp}\), do the following.

- Let \(\text{src2}\) be the double-precision floating-point value in doubleword element 0 of VSR[XT].
- Let \(\text{src3}\) be the double-precision floating-point value in doubleword element 0 of VSR[XB].

\(\text{src1}\) is multiplied\(^1\) by \(\text{src3}\), producing a product having unbounded range and precision.

See part 1 of Table 84.

\(\text{src2}\) 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.

The intermediate result is rounded to double-precision using the rounding mode specified by \(\text{RN}\).

See Table 50, “Scalar Floating-Point Intermediate Result Handling,” on page 515.

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 undefined.

\(\text{FPRF}\) 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.

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 85, “Scalar Floating-Point Final Result with Negation,” on page 611.

Special Registers Altered

\[\begin{array}{llllllll}
\text{FPRF} & \text{FR} & \text{FI} & \text{OX} & \text{UX} & \text{XX} & \text{VXSNAN} & \text{VXISI} & \text{VXIMZ}
\end{array}\]

---

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 `xsnmadd(a|m)dp`

src1 = VSR[XA]

<table>
<thead>
<tr>
<th>0</th>
<th>64</th>
<th>127</th>
</tr>
</thead>
<tbody>
<tr>
<td>DP</td>
<td>unused</td>
<td></td>
</tr>
</tbody>
</table>

src2 = `xsnmaddadp` ? VSR[XT] : VSR[XB]

<table>
<thead>
<tr>
<th>0</th>
<th>64</th>
<th>127</th>
</tr>
</thead>
<tbody>
<tr>
<td>DP</td>
<td>unused</td>
<td></td>
</tr>
</tbody>
</table>

src3 = `xsnmaddadp` ? VSR[XB] : VSR[XT]

<table>
<thead>
<tr>
<th>0</th>
<th>64</th>
<th>127</th>
</tr>
</thead>
<tbody>
<tr>
<td>DP</td>
<td>unused</td>
<td></td>
</tr>
</tbody>
</table>

tgt = VSR[XT]

<table>
<thead>
<tr>
<th>0</th>
<th>64</th>
<th>127</th>
</tr>
</thead>
<tbody>
<tr>
<td>DP</td>
<td>undefined</td>
<td></td>
</tr>
</tbody>
</table>
### Table 84. Actions for xsnmadd(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</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>–NZF</td>
<td>p ← –Infinity</td>
<td>p ← M(src1,src3)</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← –Infinity</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 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 ← –Infinity</td>
<td>p ← src3</td>
<td>p ← Q(src3) vxsnan_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 ← +Infinity</td>
<td>p ← src3</td>
<td>p ← Q(src3) vxsnan_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) vxsnan_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) 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</td>
</tr>
<tr>
<td>SNaN</td>
<td>p ← Q(src1) vxsnan_flag ← 1</td>
<td>p ← Q(src1) vximz_flag ← 1</td>
<td>p ← Q(src1) vximz_flag ← 1</td>
<td>p ← Q(src1) vximz_flag ← 1</td>
<td>p ← Q(src1) vximz_flag ← 1</td>
<td>p ← Q(src1) vximz_flag ← 1</td>
<td>p ← Q(src1) vximz_flag ← 1</td>
<td>p ← 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**: For **xsnmaddadp**, the double-precision floating-point value in doubleword element 0 of VSR[XT].
- **src3**: For **xsnmaddadp**, the double-precision floating-point value in doubleword element 0 of VSR[XB].
- For **xsnmaddmdp**, 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.
- **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>VE</th>
<th>OE</th>
<th>UE</th>
<th>XE</th>
<th>XE</th>
<th>vxsnan_flag</th>
<th>vximz_flag</th>
<th>vxisi_flag</th>
<th>Returned Results</th>
<th>Status Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Special</strong></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>T(N((r))), FPFR= ClassFP((r)), Fl=0, Fr=0</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>T((r)), FPFR= ClassFP((r)), Fl=0, Fr=0, fx(VXSI)</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>T((r)), FPFR= ClassFP((r)), Fl=0, Fr=0, fx(VXIMZ)</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>T((r)), FPFR= ClassFP((r)), Fl=0, Fr=0, fx(VXSNAN)</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>fx(VXSI), error()</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>fx(VXIMZ), error()</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>fx(VXSNAN), error()</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>fx(VXSNAN), fx(VXIMZ), error()</td>
<td>-</td>
</tr>
<tr>
<td><strong>Normal</strong></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>no</td>
<td>-</td>
<td>-</td>
<td>T(N((r))), FPFR= ClassFP((N((r)))), Fl=0, Fr=0</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>yes</td>
<td>no</td>
<td>-</td>
<td>T(N((r))), FPFR= ClassFP((N((r)))), Fl=1, Fr=0, fx(XX)</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>yes</td>
<td>yes</td>
<td>-</td>
<td>T(N((r))), FPFR= ClassFP((N((r)))), Fl=1, Fr=1, fx(XX)</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>yes</td>
<td>no</td>
<td>-</td>
<td>T(N((r))), FPFR= ClassFP((N((r)))), Fl=1, Fr=0, fx(XX), error()</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>yes</td>
<td>yes</td>
<td>-</td>
<td>T(N((r))), FPFR= ClassFP((N((r)))), Fl=1, Fr=1, fx(XX), error()</td>
<td>-</td>
</tr>
<tr>
<td><strong>Overflow</strong></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>T(N((r))), FPFR= ClassFP((N((r)))), Fl=1, Fl=7, fx(OX), fx(XX)</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>T(N((r))), FPFR= ClassFP((N((r)))), Fl=1, Fl=7, fx(OX), fx(XX), error()</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>T(N((r))), FPFR= ClassFP((N((r)))), Fl=0, Fr=0, fx(OX), error()</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>T(N((r))), FPFR= ClassFP((N((r)))), Fl=1, Fl=7, fx(OX), fx(XX), error()</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>T(N((r))), FPFR= ClassFP((N((r)))), Fl=1, Fl=7, fx(OX), fx(XX), error()</td>
<td>-</td>
</tr>
</tbody>
</table>

### Explanation:

- The results do not depend on this condition.
- ClassFP(\(x\)) = \(x\) if \(x \neq 0\). \(x\) is set to 1.
- \(\beta\) = 2^{1536} for double-precision and \(\beta = 2^{192}\) for single-precision.
- \(q\) = The value defined in Table 50, “Scalar Floating-Point Intermediate Result Handling,” on page 515, significand rounded to the target precision, unbounded exponent range.
- \(r\) = The value defined in Table 50, “Scalar Floating-Point Intermediate Result Handling,” on page 515, 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.
- FI = Floating-Point Fraction Inexact status flag, FPSCRFI. This status flag is nonsticky.
- FR = Floating-Point Fraction Rounded status flag, FPSCRFR.
- OX = Floating-Point Overflow Exception status flag, FPSCRUX.
- error() = The system error handler is invoked for the trap-enabled exception if the FE0 and FE1 bits in the Machine State Register are set to any mode other than the ignore-exception mode.
- N(\(x\)) = The value \(x\) 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.
- UX = Floating-Point Underflow Exception status flag, FPSCRXU.
- VXSNAN = Floating-Point Invalid Operation Exception (SNaN) status flag, FPSCRVXSNAN.
- VXIMZ = Floating-Point Invalid Operation Exception (Infinity × Zero) status flag, FPSCRVXIMZ.
- VXSI = Floating-Point Invalid Operation Exception (Infinity – Infinity) status flag, FPSCRVXSI.
- XX = Floating-Point Inexact Exception status flag, FPSCRXX. The flag is a sticky version of FPSCRXX. When FPSCRXX is set to a new value, the new value of FPSCRXX is set to the result of ORing the old value of FPSCRXX with the new value of FPSCRXX.

---

Table 85. Scalar Floating-Point Final Result with Negation
### Returned Results and Status Setting

<table>
<thead>
<tr>
<th>Case</th>
<th>VE</th>
<th>OE</th>
<th>UE</th>
<th>ZE</th>
<th>XE</th>
<th>vxsnan_flag</th>
<th>vximz_flag</th>
<th>vxisi_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></td>
<td></td>
<td></td>
<td></td>
<td>T(N(r)), FPRF=ClassFP(N(r)), FI=0, FR=0</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>yes</td>
<td>no</td>
<td></td>
<td>T(N(r)), FPRF=ClassFP(N(r)), FI=1, FR=0, fx(UX), fx(XX)</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>yes</td>
<td>yes</td>
<td></td>
<td>T(N(r)), FPRF=ClassFP(N(r)), FI=1, FR=1, fx(UX), fx(XX)</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>yes</td>
<td>yes</td>
<td></td>
<td>T(N(r)), FPRF=ClassFP(N(r)), FI=1, FR=0, fx(UX), fx(XX), error()</td>
</tr>
<tr>
<td>-</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>yes</td>
<td>no</td>
<td></td>
<td>T(N(r)), FPRF=ClassFP(N(r)), FI=1, FR=0, fx(UX), fx(XX), error()</td>
</tr>
<tr>
<td>-</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>T(N(r)), FPRF=ClassFP(N(r)), FI=1, FR=0, fx(UX), fx(XX), error()</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>T(N(r)), FPRF=ClassFP(N(r)), FI=1, FR=0, fx(UX), fx(XX), error()</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>T(N(r)), FPRF=ClassFP(N(r)), FI=1, FR=0, fx(UX), fx(XX), error()</td>
</tr>
</tbody>
</table>

### Explanation:

The results do not depend on this condition.

- **ClassFP(x)**: Classifies the floating-point value x as defined in Table 2, “Floating-Point Result Flags,” on page 371.

- **fx(x)**: FX is set to 1 if x=0. x is set to 1.

- **\(\beta\)**: Wrap adjust, where \(\beta = 2^{1026}\) for double-precision and \(\beta = 2^{192}\) for single-precision.

- **q**: The value defined in Table 50, “Scalar Floating-Point Intermediate Result Handling,” on page 515, significand rounded to the target precision, unbounded exponent range.

- **r**: The value defined in Table 50, “Scalar Floating-Point Intermediate Result Handling,” on page 515, 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.

- **FI**: Floating-Point Fraction Inexact status flag, FPSCR\(\text{FI}\). This status flag is nonsticky.

- **FR**: Floating-Point Fraction Rounded status flag, FPSCR\(\text{FR}\).

- **OX**: Floating-Point Overflow Exception status flag, FPSCR\(\text{OX}\).

- **error()**: The system error handler is invoked for the trap-enabled exception if the FE0 and FE1 bits in the Machine State Register are set to any mode other than the ignore-exception mode.

- **N(x)**: The value x is is nated 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. The contents of the remaining elements of VSR\[XT\] are undefined.

- **UX**: Floating-Point Underflow Exception status flag, FPSCR\(\text{UX}\).

- **VXSNAN**: Floating-Point Invalid Operation Exception (SNaN) status flag, FPSCR\(\text{VXSNAN}\).

- **VXIMZ**: Floating-Point Invalid Operation Exception (Infinity \* Zero) status flag, FPSCR\(\text{VXIMZ}\).

- **VXISI**: Floating-Point Invalid Operation Exception (Infinity – Infinity) status flag, FPSCR\(\text{VXISI}\).

- **XX**: Floating-Point Inexact Exception status flag, FPSCR\(\text{XX}\). The flag is a sticky version of FPSCR\(\text{FI}\). When FPSCR\(\text{FI}\) is set to a new value, the new value of FPSCR\(\text{XX}\) is set to the result of ORing the old value of FPSCR\(\text{XX}\) with the new value of FPSCR\(\text{FI}\).
VSX Scalar Negative Multiply-Add Single-Precision XX3-form

**xsnmaddasp** XT,XA,XB

<table>
<thead>
<tr>
<th>60</th>
<th>T</th>
<th>A</th>
<th>B</th>
<th>129</th>
<th>vex_flag</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>0b0</td>
</tr>
</tbody>
</table>

**xsnmaddmsp** XT,XA,XB

<table>
<thead>
<tr>
<th>60</th>
<th>T</th>
<th>A</th>
<th>B</th>
<th>137</th>
<th>vex_flag</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>0b0</td>
</tr>
</tbody>
</table>

reset_xflags();

if "xsnmaddasp" then do
  src1 <- VSR[32×AX+A].dword[0]
  src2 <- VSR[32×TX+T].dword[0]
  src3 <- VSR[32×BX+B].dword[0]
end

if "xsnmaddmsp" then do
  src1 <- VSR[32×AX+A].dword[0]
  src2 <- VSR[32×BX+B].dword[0]
  src3 <- VSR[32×TX+T].dword[0]
end

\[ v \leftarrow \text{MultiplyAddDP}(src1, src3, src2) \]
\[ \text{result} \leftarrow -\text{NegateSP}(\text{RoundToSP}(RN, v)) \]

if (vxsnan_flag) then SetFX(VXSNAN)
if (vximz_flag) then SetFX(VXIMZ)
if (vxisi_flag) then SetFX(VXISI)
if (ox_flag) then SetFX(OX)
if (ux_flag) then SetFX(UX)
if (xx_flag) then SetFX(XX)

vex_flag <- VE & (vxsnan_flag | vximz_flag | vxisi_flag)

if ~vex_flag then do
  VSR[32×TX+T].dword[0] <- ConvertToSP(result)
  VSR[32×TX+T].dword[1] <- 0xUUUU_UUUU_UUUU_UUUU
  FPRF <- ClassSP(result)
  FR <- inc_flag
  FI <- xx_flag
else do
  FR <- 0b0
  FI <- 0b0
end

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 **xsnmaddasp**, 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 **xsnmaddmsp**, 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 by src3, producing a product having unbounded range and precision.

See part 1 of Table 86, “Actions for xsnmadd(a|m)sp,” on page 615.

src2 is added to the product, producing a sum having unbounded range and precision.

The sum is normalized.

See part 2 of Table 86, “Actions for xsnmadd(a|m)sp,” on page 615.

The intermediate result is rounded to single-precision using the rounding mode specified by RN.

See Table 50, “Scalar Floating-Point Intermediate Result Handling,” on page 515.

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 undefined.

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 85, “Scalar Floating-Point Final Result with Negation,” on page 611.

---

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.
**Special Registers Altered**

<table>
<thead>
<tr>
<th>FPRF</th>
<th>FR</th>
<th>FI</th>
<th>FX</th>
<th>OX</th>
<th>UX</th>
<th>XX</th>
</tr>
</thead>
<tbody>
<tr>
<td>VXSNAN</td>
<td>VXISI</td>
<td>VXIMZ</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**VSR Data Layout for xsnmadd(a|m)sp**

src1 = VSR[XA]

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>DP</td>
<td>unused</td>
</tr>
</tbody>
</table>

src2 = $xsnmadda(dp|sp)$ ? VSR[XT] : VSR[XB]

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>DP</td>
<td>unused</td>
</tr>
</tbody>
</table>

src3 = $xsnmadda(dp|sp)$ ? VSR[XB] : VSR[XT]

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>DP</td>
<td>unused</td>
</tr>
</tbody>
</table>

tgt = VSR[XT]

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>DP</td>
<td>undefined</td>
</tr>
</tbody>
</table>
Table 86. Actions for xsnmadd(a|m)sp

<table>
<thead>
<tr>
<th>src3</th>
<th>src1</th>
<th>src2</th>
</tr>
</thead>
<tbody>
<tr>
<td>–Infinity</td>
<td>–Infinity</td>
<td>–Infinity</td>
</tr>
<tr>
<td>p ← +Infinity</td>
<td>p ← +Infinity</td>
<td>p ← +Infinity</td>
</tr>
<tr>
<td>p ← dQNaN</td>
<td>p ← src1</td>
<td>p ← dQNaN</td>
</tr>
<tr>
<td>vximz_flag ← 1</td>
<td>p ← –Infinity</td>
<td>p ← –Infinity</td>
</tr>
<tr>
<td>+Zero</td>
<td>+Zero</td>
<td>+Zero</td>
</tr>
<tr>
<td>p ← +Zero</td>
<td>p ← +Zero</td>
<td>p ← +Zero</td>
</tr>
<tr>
<td>p ← +Zero</td>
<td>p ← +Zero</td>
<td>p ← +Zero</td>
</tr>
<tr>
<td>–Zero</td>
<td>–Zero</td>
<td>–Zero</td>
</tr>
<tr>
<td>p ← +Infinity</td>
<td>p ← +Infinity</td>
<td>p ← +Infinity</td>
</tr>
<tr>
<td>p ← dQNaN</td>
<td>p ← src1</td>
<td>p ← dQNaN</td>
</tr>
<tr>
<td>vximz_flag ← 1</td>
<td>p ← –Infinity</td>
<td>p ← –Infinity</td>
</tr>
<tr>
<td>QNaN</td>
<td>QNaN</td>
<td>QNaN</td>
</tr>
<tr>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← src1</td>
</tr>
<tr>
<td>QNaN</td>
<td>QNaN</td>
<td>QNaN</td>
</tr>
<tr>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← src1</td>
</tr>
<tr>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← src1</td>
</tr>
<tr>
<td>SNaN</td>
<td>SNaN</td>
<td>SNaN</td>
</tr>
<tr>
<td>p ← Q(src1)</td>
<td>p ← Q(src1)</td>
<td>p ← Q(src1)</td>
</tr>
<tr>
<td>vximz_flag ← 1</td>
<td>p ← Q(src1)</td>
<td>p ← Q(src1)</td>
</tr>
<tr>
<td>+Zero</td>
<td>+Zero</td>
<td>+Zero</td>
</tr>
<tr>
<td>p ← +Zero</td>
<td>p ← +Zero</td>
<td>p ← +Zero</td>
</tr>
<tr>
<td>p ← +Zero</td>
<td>p ← +Zero</td>
<td>p ← +Zero</td>
</tr>
</tbody>
</table>

**Explanation:**

src1 The double-precision floating-point value in doubleword element 0 of VSR[XA].

src2 For xsnmaddsp, the double-precision floating-point value in doubleword element 0 of VSR[XT].
For xsnmaddmsp, the double-precision floating-point value in doubleword element 0 of VSR[XB].

src3 For xsnmaddsp, the double-precision floating-point value in doubleword element 0 of VSR[XB].
For xsnmaddmsp, the double-precision floating-point value in doubleword element 0 of VSR[XT].

dQNaN Default quiet NaN (0x7FFP8_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.

Table 86. Actions for xsnmadd(a|m)sp

Chapter 7. Vector-Scalar Floating-Point Operations 615
VSX Scalar Negative Multiply-Add Quad-Precision [using round to Odd] X-form

xsnmaddqp VRT, VRA, VRB (RO=0)

xsnmaddqpo VRT, VRA, VRB (RO=1)

<table>
<thead>
<tr>
<th>63</th>
<th>6</th>
<th>51</th>
<th>16</th>
<th>21</th>
<th>452</th>
<th>21</th>
</tr>
</thead>
</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, src2)
rvd <- bfp_NEGATE(bfp_ROUND_TO_BFP128(RO, FPSCR.RN, v))
result <- bfp_CONVERT_TO_BFP128(rvd)

if vxsnan_flag then SetFX(FPSCR.VXSNAN)
if vximz_flag then SetFX(FPSCR.VXIMZ)
if vxisi_flag then SetFX(FPSCR.VXISI)
if vxli_flag then SetFX(FPSCR.VXLI)
if vxlo_flag then SetFX(FPSCR.VXLO)
if vx_mz_flag then SetFX(FPSCR.VX_MZ)

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.FPFR <- 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 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 69. "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 69. "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
significant 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.

See Table 50, "Scalar Floating-Point Intermediate
Result Handling," on page 515.

The result is negated and 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.

1. The quad-precision default Quiet NaN is the value, 0x7FFFF_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 51, “VSX Scalar Floating-Point Final Result,” on page 516.

Special Registers Altered:
- FPRF
- FR
- FI
- FX, VXSNAN, VXIMZ, VXISI, OX, UX, XX

VSR Data Layout for xsnmaddqp[o]

| VSR[VRB+32] | src3          |
| VSR[VRT+32] | src2          |
| VSR[VRB+32] | src1          |
| VSR[VRT+32] | tgt           |
### Table 87. Actions for xsnmaddq[p][o]

<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>p – Zero</td>
<td>p + Zero</td>
<td>p + Infinity</td>
<td>p + Infinity</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+NZF</td>
<td>p + NewNZ(src3)</td>
<td>p + QNaN</td>
<td>p – Zero</td>
<td>p + Zero</td>
<td>p + Infinity</td>
<td>p + Infinity</td>
<td></td>
<td></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>v + QNaN</td>
<td>v – Zero</td>
<td>v + Zero</td>
<td>v + Infinity</td>
<td>v + Infinity</td>
<td></td>
<td></td>
</tr>
<tr>
<td>–NZF</td>
<td>v + Add(p,src2)</td>
<td>v + p</td>
<td>v + Add(p,src2)</td>
<td>v + p</td>
<td>v + Add(p,src2)</td>
<td>v + Add(p,src2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>–Zero</td>
<td>v + src2</td>
<td>v + p</td>
<td>v + p</td>
<td>v + src2</td>
<td>v + p</td>
<td>v + src2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+Zero</td>
<td>v + Newrez</td>
<td>v + src2</td>
<td>v + p</td>
<td>v + src2</td>
<td>v + p</td>
<td>v + src2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+NZF</td>
<td>v + Add(p,src2)</td>
<td>v + p</td>
<td>v + p</td>
<td>v + src2</td>
<td>v + p</td>
<td>v + src2</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[VRT+32].
- **src3**: The quad-precision floating-point value in VSR[VRA+32].
- **QNaN**: Default quiet NaN (0x7FFFFF_8000_0000_0000_0000_0000_0000_0000).
- **NZF**: Nonzero finite number.
- **Rezv**: 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 (Rezv).
- **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.
**VSX Scalar Negative Multiply-Subtract**

**Double-Precision XX3-form**

### xsnmsubadp

<table>
<thead>
<tr>
<th>60</th>
<th>6</th>
<th>A</th>
<th>B</th>
<th>177</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>T</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

### xsnmsubmdp

<table>
<thead>
<tr>
<th>60</th>
<th>6</th>
<th>A</th>
<th>B</th>
<th>185</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>T</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

- **T**: 
  - TX || T
- **A**: 
  - AX || A
- **B**: 
  - BX || B
- **reset_xflags()**: 
  - src1 \(\Leftarrow VSR[XA]{0:63}\)
  - src2 \(\Leftarrow VSR[XT]{0:63}\)
  - src3 \(\Leftarrow VSR[XB]{0:63}\)
  - \(v[{\text{0:inf}}] \Leftarrow \text{MultiplyAddDP(src1,src3,src2)}\)
  - \(\text{result}[0:63] \Leftarrow \text{NegateDP(RoundToDP(RN,v))}\)
  - if(vxsnan_flag) then SetFX(VXSNAN)
  - if(vximz_flag) then SetFX(VXIMZ)
  - if(vxisi_flag) then SetFX(VXISI)
  - if(ox_flag) then SetFX(OX)
  - if(ux_flag) then SetFX(UX)
  - if(xx_flag) then SetFX(XX)
  - vex_flag \(\Leftarrow \text{VE} \& (\text{vxsnan_flag} \| \text{vximz_flag} \| \text{vxisi_flag})\)

- **src1** is multiplied\(^1\) by **src3**, producing a product having unbounded range and precision.

- **src2** is negated and added\(^2\) to the product, producing a sum having unbounded range and precision.

- The sum is normalized\(^3\).

**Special Registers Altered**

- FPRF
- FR
- FI
- FX
- 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 xsnmsub(a|m)dp

<table>
<thead>
<tr>
<th>src1 = VSR[XA]</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>DP</td>
<td>unused</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>src2 = xsnmsubadp ? VSR[XT] : VSR[XB]</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>DP</td>
<td>unused</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>src3 = xsnmsubadp ? VSR[XB] : VSR[XT]</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>DP</td>
<td>unused</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>tgt = VSR[XT]</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>DP</td>
<td>undefined</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0</th>
<th>64</th>
<th>127</th>
</tr>
</thead>
</table>
### Table 88. Actions for `xsnmsub(a|m)dp`

<table>
<thead>
<tr>
<th>src</th>
<th>Multiply</th>
<th>Subtract</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Infinity</td>
<td>( p = +\text{Infinity} )</td>
<td>( v = -\text{Infinity} )</td>
</tr>
<tr>
<td>-NZF</td>
<td>( p = +\text{Infinity} )</td>
<td>( v = -\text{Infinity} )</td>
</tr>
<tr>
<td>-Zero</td>
<td>( p = +\text{Zero} )</td>
<td>( v = -\text{Infinity} )</td>
</tr>
<tr>
<td>+Zero</td>
<td>( p = -\text{Zero} )</td>
<td>( v = -\text{Infinity} )</td>
</tr>
<tr>
<td>+NZF</td>
<td>( p = +\text{Zero} )</td>
<td>( v = -\text{Infinity} )</td>
</tr>
<tr>
<td>+Infinity</td>
<td>( p = +\text{Zero} )</td>
<td>( v = -\text{Infinity} )</td>
</tr>
<tr>
<td>QNaN</td>
<td>( p = Q(src1) )</td>
<td>( v = -\text{Infinity} )</td>
</tr>
<tr>
<td>SNaN</td>
<td>( p = Q(src1) )</td>
<td>( v = -\text{Infinity} )</td>
</tr>
</tbody>
</table>

#### Explanation:

- **src1** The double-precision floating-point value in doubleword element 0 of VSR[XT].
- **src2** The intermediate product having unbounded range and precision.
- **src3** The intermediate result having unbounded range and precision.
- **dQNaN** Default quiet NaN \( (0x7FPB_0000_0000_0000) \).
- **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: \( x = y \) when 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.
- **v** The intermediate result having unbounded range and precision.
### VSX Scalar Negative Multiply-Subtract Single-Precision XX3-form

**xsnmsubasp**  
\[ XT,XA,XB \]

- **0**
- **6**
- **11**
- **15**
- **145**
- **Kb1**
- **Kb2**

**xsnmsubmsp**  
\[ XT,XA,XB \]

- **0**
- **6**
- **11**
- **15**
- **153**
- **Kb1**
- **Kb2**

**reset_xflags()**

If "xsnmsubasp" then do

1. `src1` \( \leftarrow \text{VSR}[32\times AX+A].\text{dword}[0] \)
2. `src2` \( \leftarrow \text{VSR}[32\times TX+T].\text{dword}[0] \)
3. `src3` \( \leftarrow \text{VSR}[32\times BX+B].\text{dword}[0] \)

End

If "xsnmsubmsp" then do

1. `src1` \( \leftarrow \text{VSR}[32\times AX+A].\text{dword}[0] \)
2. `src2` \( \leftarrow \text{VSR}[32\times BX+B].\text{dword}[0] \)
3. `src3` \( \leftarrow \text{VSR}[32\times TX+T].\text{dword}[0] \)

End

\[
\begin{align*}
v & \leftarrow \text{MultiplyAddDP}(src1,src3,\text{NegateDP}(src2)) \\
\text{result} & \leftarrow \text{NegateSP}\left(\text{RoundToSP}(\text{RN},v)\right)
\end{align*}
\]

- If `vxsnan_flag` then SetFX(VXSNAN)
- If `vximz_flag` then SetFX(VXIMZ)
- If `vxisi_flag` then SetFX(VXISI)
- If `ox_flag` then SetFX(OX)
- If `ux_flag` then SetFX(UX)
- If `xx_flag` then SetFX(XX)

\[vex_flag \leftarrow \text{VE } \& \text{vxsnan_flag } \text{| vximz_flag } \text{| vxisi_flag}\]

If `~vex_flag` then do

1. \(\text{VSR}[32\times TX+T].\text{dword}[0] \leftarrow \text{ConvertSPtoSP64}(\text{result})\)
2. \(\text{VSR}[32\times TX+T].\text{dword}[1] \leftarrow 0x\text{UUUU}_\text{UUUU}_\text{UUUU}_\text{UUUU}\)
3. \(\text{FPRF} \leftarrow \text{ClassSP(\text{result})}\)
4. \(\text{FR} \leftarrow \text{inc_flag}\)
5. \(\text{FI} \leftarrow xx_flag\)

Else do

1. \(\text{FR} \leftarrow 0\text{b}0\)
2. \(\text{FI} \leftarrow 0\text{b}0\)

End

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 xsnmsubasp**, do the following.

- Let `src1` be the double-precision floating-point value in doubleword element 0 of VSR[XT].
- 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[XT].

**For xsnmsubmsp**, do the following.

- Let `src1` be the double-precision floating-point value in doubleword element 0 of VSR[XT].
- 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[XT].

`src1` is multiplied\(^1\) by `src3`, producing a product having unbounded range and precision.

See part 1 of Table 89, “Actions for xsnmsub(a|m)sp,” on page 624.

`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 89, “Actions for xsnmsub(a|m)sp,” on page 624.

The intermediate result is rounded to single-precision using the rounding mode specified by RN.

See Table 50, “Scalar Floating-Point Intermediate Result Handling,” on page 515.

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 undefined.

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 85, “Scalar Floating-Point Final Result with Negation,” on page 611.

---

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.
### Special Registers Altered

<table>
<thead>
<tr>
<th></th>
<th>FPRF</th>
<th>FR</th>
<th>FI</th>
<th>FX</th>
<th>OX</th>
<th>UX</th>
<th>XX</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>VXSNAN</td>
<td>VXISI</td>
<td>VXIMZ</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### VSR Data Layout for xsnmsub(a|m)sp

<table>
<thead>
<tr>
<th>src1 = VSR[XA]</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>DP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>unused</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>src2 = xsnmsubasp ? VSR[XT] : VSR[XB]</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>DP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>unused</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>src3 = xsnmsubasp ? VSR[XB] : VSR[XT]</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>DP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>unused</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>tgt = VSR[XT]</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>DP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>undefined</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0</th>
<th>64</th>
<th>127</th>
</tr>
</thead>
</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 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) vxnan_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) vxnan_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 ← –Infinity</td>
<td>p ← src3</td>
<td>p ← Q(src3) vxnan_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 ← +Infinity</td>
<td>p ← src3</td>
<td>p ← Q(src3) vxnan_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) vxnan_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) vxnan_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 ← Q(src1) vxnan_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) vxnan_flag ← 1</td>
<td>p ← Q(src1) vxnan_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 vxsl_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) vxnan_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) vxnan_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) vxnan_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) vxnan_flag ← 1</td>
</tr>
<tr>
<td>+NZF</td>
<td>v ← +Infinity</td>
<td>v ← +src2</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← src2</td>
<td>v ← Q(src2) vxnan_flag ← 1</td>
</tr>
<tr>
<td>+Infinity</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 ← src2</td>
<td>v ← Q(src2) vxnan_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 ← 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 VSR[XA].dword[0].
- **src2**: For **xsnmsubasp**, the double-precision floating-point value in VSR[XT].dword[0].
- **src3**: For **xsnmsubasp**, 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.
- **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.

### Table 89: Actions for xsnmsub(a|m|sp)
Chapter 7. Vector-Scalar Floating-Point Operations

VSX Scalar Negative Multiply-Subtract Quad-Precision [using round to Odd] X-form

<table>
<thead>
<tr>
<th>xsnmsubqp</th>
<th>VRT, VRA, VRB</th>
<th>(RO=0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>xsnmsubqpo</td>
<td>VRT, VRA, VRB</td>
<td>(RO=1)</td>
</tr>
</tbody>
</table>

63 6 11 16 21 31 484 484

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_NEGATE(bfp_ROUND_TO_BFP128_RO, FPSCR.RN, v)
result ← bfp_CONVERT_TO_BFP128(rnd)

if(vxsnan_flag) then SetFX(FPSCR.VXSNAN)
if(vximz_flag) then SetFX(FPSCR.VXIMZ)
if(vxisi_flag) then SetFX(FPSCR.VXISI)
if(ox_flag) then SetFX(FPSCR.OX)
if(ux_flag) then SetFX(FPSCR.UX)
if(xx_flag) then SetFX(FPSCR.XX)

vx_flag ← vxsnan_flag | vximz_flag | vxisi_flag
ex_flag ← FPSCR.XE & vx_flag
if ex_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.

1. The quad-precision default Quiet NaN is the value, 0x7FFF_8000_0000_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 51, “VSX Scalar Floating-Point Final Result,” on page 516.

**Special Registers Altered:**
- \( FPRF \)
- \( FR \)
- \( FI \)
- \( FX \) \( VXSNAN \)
- \( VXIMZ \)
- \( VXISI \)
- \( OX \)
- \( UX \)
- \( XX \)

### VSR Data Layout for xsnmsubqp[o]

<table>
<thead>
<tr>
<th>VSR(VRA+32)</th>
<th>src1</th>
</tr>
</thead>
<tbody>
<tr>
<td>VSR(VRT+32)</td>
<td>src2</td>
</tr>
<tr>
<td>VSR(VRB+32)</td>
<td>src3</td>
</tr>
<tr>
<td>VSR(VRT+32)</td>
<td>tgt</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 + QNaN</td>
<td>p + QNaN</td>
<td>p + Infinity</td>
<td>p + Infinity</td>
<td>p + Infinity</td>
<td>p + Infinity</td>
<td></td>
</tr>
<tr>
<td>-NZF</td>
<td>p + QNaN</td>
<td>p + QNaN</td>
<td>p + QNaN</td>
<td>p + QNaN</td>
<td>p + QNaN</td>
<td>p + QNaN</td>
<td>p + QNaN</td>
<td></td>
</tr>
<tr>
<td>-Zero</td>
<td>p + QNaN</td>
<td>p + QNaN</td>
<td>p + QNaN</td>
<td>p + QNaN</td>
<td>p + QNaN</td>
<td>p + QNaN</td>
<td>p + QNaN</td>
<td></td>
</tr>
<tr>
<td>+Zero</td>
<td>p + QNaN</td>
<td>p + QNaN</td>
<td>p + QNaN</td>
<td>p + QNaN</td>
<td>p + QNaN</td>
<td>p + QNaN</td>
<td>p + QNaN</td>
<td></td>
</tr>
<tr>
<td>+NZF</td>
<td>p + QNaN</td>
<td>p + QNaN</td>
<td>p + QNaN</td>
<td>p + QNaN</td>
<td>p + QNaN</td>
<td>p + QNaN</td>
<td>p + QNaN</td>
<td></td>
</tr>
<tr>
<td>+Infinity</td>
<td>p + QNaN</td>
<td>p + QNaN</td>
<td>p + QNaN</td>
<td>p + QNaN</td>
<td>p + QNaN</td>
<td>p + QNaN</td>
<td>p + QNaN</td>
<td></td>
</tr>
<tr>
<td>QNaN</td>
<td>p + QNaN</td>
<td>p + QNaN</td>
<td>p + QNaN</td>
<td>p + QNaN</td>
<td>p + QNaN</td>
<td>p + QNaN</td>
<td>p + QNaN</td>
<td></td>
</tr>
<tr>
<td>SNaN</td>
<td>p + QNaN</td>
<td>p + QNaN</td>
<td>p + QNaN</td>
<td>p + QNaN</td>
<td>p + QNaN</td>
<td>p + QNaN</td>
<td>p + QNaN</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>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></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></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></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></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></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></td>
</tr>
<tr>
<td>QNaN &amp; src1 is a NaN</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></td>
</tr>
<tr>
<td>QNaN &amp; src1 not a NaN</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></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_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 90. Actions for xsnmsubqp[0] |
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>30</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>T</td>
<td>///</td>
<td>B</td>
<td>73</td>
<td>B'X</td>
</tr>
</tbody>
</table>

- \( \text{XT} \leftarrow TX || T \)
- \( \text{XB} \leftarrow BX || B \)
- \( \text{reset} \_\text{flags}() \)
- \( \text{result}(0:63) \leftarrow \text{RoundToDPIntegerNearAway}(\text{VSR}[\text{XB}](0:63)) \)
- if(\( \text{vxsnan} \_\text{flag} \)) then \( \text{SetFX}(\text{VXSNAN}) \)
- \( FR \leftarrow 0b0 \)
- \( FI \leftarrow 0b0 \)
- \( \text{vx}_{\text{flag}} \leftarrow \text{VE} \& \text{vxsnan} \_\text{flag} \)
- if(\( \sim \text{vx}_{\text{flag}} \)) then do
  - \( \text{VSR}[\text{XT}] \leftarrow \text{result} \| 0xUUUU\_UUUU\_UUUU\_UUUU \)
  - \( \text{FPRF} \leftarrow \text{ClassFP}(\text{result}) \)
end

Let \( \text{XT} \) be the value \( 32 \times TX + T \).
Let \( \text{XB} \) be the value \( 32 \times BX + B \).

Let \( \text{src} \) be the double-precision floating-point value in doubleword element 0 of \( \text{VSR}[\text{XB}] \).

\( \text{src} \) is rounded to an integer using the rounding mode Round to Nearest Away.

The 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 undefined.

\( \text{FPRF} \) 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, \( \text{VSR}[\text{XT}] \) and \( \text{FPRF} \) are not modified, and \( FR \) and \( FI \) are set to 0.

Special Registers Altered
- \( \text{FPRF} \)
- \( FR=0b0 \)
- \( FI=0b0 \)
- \( FX \)
- \( VXSNAN \)

**VSR Data Layout for xsrdpi**

<table>
<thead>
<tr>
<th>src = VSR[XB]</th>
</tr>
</thead>
<tbody>
<tr>
<td>DP</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>tgt = VSR[XT]</th>
</tr>
</thead>
<tbody>
<tr>
<td>DP</td>
</tr>
</tbody>
</table>

**Programming Note**

This instruction can be used to operate on a single-precision source operand.
VSX Scalar Round to Double-Precision Integer exact using Current rounding mode XX2-form

$$\text{xsrdpic} \ XT, XB$$

|          | 60 | 6 | 11 || 16 | 21 | 107 | B | 8 |
|----------|----|---|-----||----|----|-----|---|---|
| XT       | ← TX || T                       |
| XB       | ← BX || B                       |
| reset_xflags() | ← VSR[XB][0:63]               |
| src      | ← VSR[XB][0:63]               |
| if(RN=0000) then result[0:63] ← RoundToDPIntegerNearEven(src) |
| if(RN=0001) then result[0:63] ← RoundToDPIntegerTrunc(src) |
| if(RN=0010) then result[0:63] ← RoundToDPIntegerCeil(src) |
| if(RN=0011) then result[0:63] ← RoundToDPIntegerFloor(src) |
| if(vxsnan_flag) then SetFX(VXSNAN) |
| if(xx_flag) then SetFX(XX) |
| vex_flag  | ← VE & vxsnan_flag            |
| if(¬vex_flag) then do          |
| VSR[XT]  | ← result || 0xUUUU_UUUU_UUUU_UUUU |
| FPRF     | ← ClassDP(result)            |
| FR       | ← inc_flag                   |
| FI       | ← xx_flag                    |
| end      |                               |
| else do  |                               |
| FR       | ← 0b0                        |
| 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].

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 undefined.

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 FX VXSNAN

Programming Note

This instruction can be used to operate on a single-precision source operand.
VSX Scalar Round to Double-Precision Integer using round toward -Infinity XX2-form

xsrdpim \(XT, XB\)

<table>
<thead>
<tr>
<th>60</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>121</th>
<th>6X/TX</th>
</tr>
</thead>
<tbody>
<tr>
<td>XT</td>
<td>← TX</td>
<td></td>
<td>T</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XB</td>
<td>← BX</td>
<td></td>
<td>B</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

reset_xflags()

result{0:63} \(\leftarrow\) RoundToDPIntegerFloor(VSR[XB]{0:63})

if(vxsnan_flag) then SetFX(VXSNAN)

FR \(\leftarrow\) 0b0

FI \(\leftarrow\) 0b0

vex_flag \(\leftarrow\) VE & vxsnan_flag

if( ~vex_flag ) then do

VSR[XT] \(\leftarrow\) result || 0xUUUU_UUUU_UUUU_UUUU

FPRF \(\leftarrow\) ClassDP(result)

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].

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 undefined.

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

VSR Data Layout for xsrdpim

src = VSR[XB]

tgt = VSR[XT]

<table>
<thead>
<tr>
<th>DP</th>
<th>unused</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>64</td>
</tr>
<tr>
<td>64</td>
<td>127</td>
</tr>
</tbody>
</table>

Programming Note

This instruction can be used to operate on a single-precision source operand.

VSX Scalar Round to Double-Precision Integer using round toward +Infinity XX2-form

xsrdpip \(XT, XB\)

<table>
<thead>
<tr>
<th>60</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>105</th>
<th>6X/TX</th>
</tr>
</thead>
<tbody>
<tr>
<td>XT</td>
<td>← TX</td>
<td></td>
<td>T</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XB</td>
<td>← BX</td>
<td></td>
<td>B</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

reset_xflags()

result{0:63} \(\leftarrow\) RoundToDPIntegerCeil(VSR[XB]{0:63})

if(vxsnan_flag) then SetFX(VXSNAN)

FR \(\leftarrow\) 0b0

FI \(\leftarrow\) 0b0

vex_flag \(\leftarrow\) VE & vxsnan_flag

if( ~vex_flag ) then do

VSR[XT] \(\leftarrow\) result || 0xUUUU_UUUU_UUUU_UUUU

FPRF \(\leftarrow\) ClassDP(result)

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].

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 undefined.

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

VSR Data Layout for xsrdpip

src = VSR[XB]

tgt = VSR[XT]

<table>
<thead>
<tr>
<th>DP</th>
<th>unused</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>64</td>
</tr>
<tr>
<td>64</td>
<td>127</td>
</tr>
</tbody>
</table>

Programming Note

This instruction can be used to operate on a single-precision source operand.
VSX Scalar Round to Double-Precision Integer using round toward Zero XX2-form

**xsrdpiz**  
**XT, XB**

<table>
<thead>
<tr>
<th>60</th>
<th>5</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>T</td>
<td>16</td>
<td>89</td>
<td>B1</td>
<td>88</td>
<td>87</td>
<td>86</td>
<td>85</td>
<td>84</td>
</tr>
</tbody>
</table>

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 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 undefined.

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

**VSR Data Layout for xsrdpiz**

**src** = **VSR[XB]**

<table>
<thead>
<tr>
<th>DP</th>
<th>unused</th>
</tr>
</thead>
</table>

**tgt** = **VSR[XT]**

<table>
<thead>
<tr>
<th>DP</th>
<th>undefined</th>
</tr>
</thead>
</table>

**Programming Note**

This instruction can be used to operate on a single-precision source operand.
VSX Scalar Reciprocal Estimate
Double-Precision XX2-form

xsredp XT, XB

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 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,

\[
\left| \frac{1}{\text{estimate}} - \frac{1}{\text{src}} \right| \leq \frac{1}{16384}
\]

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</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>SNanN</td>
<td>QNaN(^2)</td>
<td>VXSNaN</td>
</tr>
<tr>
<td>QNaNN</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 undefined.

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
FPRF FR=0bU FI=0bU FX OX UX XX=0bU VXSNAN

VSR Data Layout for xsredp
src = VSR[XB]

tgt = VSR[XT]

<table>
<thead>
<tr>
<th>src</th>
<th>tgt</th>
</tr>
</thead>
<tbody>
<tr>
<td>DP</td>
<td>unused</td>
</tr>
<tr>
<td>undefined</td>
<td></td>
</tr>
</tbody>
</table>
VSX Scalar Reciprocal Estimate
Single-Precision XX2-form

xsresp XT,XB

<table>
<thead>
<tr>
<th></th>
<th>60</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>26</th>
<th>B</th>
<th>X</th>
<th>T</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>

reset_xflags();

src ← VSR[32×BX+B].dword[0]
v ← ReciprocalEstimateDP[src]
result ← RoundToSP(RN,v)

if(vxsnan_flag) then SetFX(VXSNAN)
if(oop_flag) then SetFX(OX)
if(oo_flag) then SetFX(UX)
if(zx_flag) then SetFX(ZX)
if(0bU) then SetFX(XX)

vex_flag ← VE & vxsnan_flag
zex_flag ← ZE & zx_flag

if( ~vex_flag & ~zex_flag ) then do
  VSR[32×TX+T].dword[0] ← ConvertSPtoSP64(result)
  VSR[32×TX+T].dword[1] ← 0xUUUU_UUUU_UUUU_UUUU
  FPRF ← ClassSP(result)
  FR ← 0bU
  FI ← 0bU
end
else do
  FR ← 0b0
  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].

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</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
undefined.

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 ZX XX=0bU
VXSNAN

VSR Data Layout for xsresp

src = VSR[XB]
tgt = VSR[XT]
### VSX Scalar Round to Quad-Precision Integer [with Inexact] Z23-form

#### xsrqpi

<table>
<thead>
<tr>
<th>R, VRT, VRB, RMC</th>
<th>(EX=0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>63</td>
</tr>
</tbody>
</table>

#### xsrqpix

<table>
<thead>
<tr>
<th>R, VRT, VRB, RMC</th>
<th>(EX=1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>63</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>0</td>
<td>00</td>
<td>–</td>
<td>Round to Nearest Away</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>–</td>
<td>reserved</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>–</td>
<td>reserved</td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>00</td>
<td>Round to Nearest Even</td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>01</td>
<td>Round towards Zero</td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>10</td>
<td>Round towards +Infinity</td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>11</td>
<td>Round towards -Infinity</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>–</td>
<td>Round to Nearest Even</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>–</td>
<td>Round towards Zero</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>–</td>
<td>Round towards +Infinity</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>–</td>
<td>Round towards -Infinity</td>
</tr>
</tbody>
</table>

#### Special Registers Altered:

- **FPRF**: Set to the class and sign of the result.
- **FR**: Set to 0, **FI** is set to 0, and **XX** is not set by an Inexact exception.
- **FX**: Set to **FPRF**.
- **VXSNA**
- **FX**: Set to 0, **FI** is set to 0, **XX** is set by an Inexact exception.
- **VXSNA**: Set to 1, and the result is the Quiet NaN corresponding to the Signalling NaN.
- **FX**: Set to **VXSNA**.
- **FX**: Set to **FR & (EX=0)**, and **XX** is set by an Inexact exception.

### Rounding Mode Table

<table>
<thead>
<tr>
<th>R</th>
<th>RMC</th>
<th>FPSCR.RN</th>
<th>Rounding Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>–</td>
<td>Round to Nearest Away</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>–</td>
<td>reserved</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>–</td>
<td>reserved</td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>00</td>
<td>Round to Nearest Even</td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>01</td>
<td>Round towards Zero</td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>10</td>
<td>Round towards +Infinity</td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>11</td>
<td>Round towards -Infinity</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>–</td>
<td>Round to Nearest Even</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>–</td>
<td>Round towards Zero</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>–</td>
<td>Round towards +Infinity</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>–</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, **VXSNA** 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.

FPFR is set to the class and sign of the result.

For **xsrqpi**, \( FR \) 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.
### VSR Data Layout for xsrqpi

<table>
<thead>
<tr>
<th>VR8+32</th>
<th>src</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR7+32</td>
<td>tgt</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>0</td>
<td>000</td>
<td>000</td>
<td>Round to Nearest Away</td>
</tr>
<tr>
<td>0</td>
<td>001</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>00</td>
<td>Round to Nearest Even</td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>01</td>
<td>Round to Zero</td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>10</td>
<td>Round to +Infinity</td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>11</td>
<td>Round to -Infinity</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>00</td>
<td>Round to Nearest Even</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>00</td>
<td>Round to Zero</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>00</td>
<td>Round to +Infinity</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>00</td>
<td>Round to -Infinity</td>
</tr>
</tbody>
</table>

Let \( src \) be the floating-point value in \( VSX[VRT+32] \) 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, with the significand truncated to double-extended-precision.

Otherwise, if \( src \) is a Quiet NaN, then the result is \( src \) with the significand truncated to double-extended-precision.

Otherwise, if \( src \) is an Infinity or a Zero, the result is \( src \).

Otherwise, \( src \) is rounded to double-extended precision (i.e., 15-bit exponent range and 64-bit significand precision) using the specified rounding mode.

See Table 50, “Scalar Floating-Point Intermediate Result Handling,” on page 515.

The result is placed into \( VSX[VRT+32] \) 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, and \( FI \) 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, \( VSX[VRT+32] \) and \( FPRF \) are not modified, and \( FR \) and \( FI \) are set to 0.

See Table 51, “VSX Scalar Floating-Point Final Result,” on page 516.
### Special Registers Altered:

| FPRT | FR  | FI  | FX  | VXSNAN | QX  | UX  | XX  |

### VSR Data Layout for xsrqpxp

| VSR[VR8+32] | src |
| VSR[VRT+32] | tgt |
VSX Scalar Round to Single-Precision
XX2-form

\[ \text{xsrsp} \quad \text{XT, XB} \]

<table>
<thead>
<tr>
<th>0</th>
<th>60</th>
<th>T</th>
<th>//</th>
<th>B</th>
<th>281</th>
<th>T 6</th>
<th>30</th>
<th>31</th>
</tr>
</thead>
</table>

reset_flags()

\[
\begin{align*}
\text{src} & \leftarrow \text{VSR}[32\times B + B].\text{dword}(0) \\
\text{result} & \leftarrow \text{RoundToSP}(\text{src}) \\
\end{align*}
\]

if (vxsnan_flag) then SetFX(VXSNAN)
if (ox_flag) then SetFX(OX)
if (ux_flag) then SetFX(UX)
if (xx_flag) then SetFX(XX)

\[
\text{vex_flag} \leftarrow \text{VE} \& \text{vxsnan_flag}
\]

if (¬vex_flag) then do

\[
\begin{align*}
\text{VSR}[32\times T + T].\text{dword}(0) & \leftarrow \text{ConvertSPtoSP64}(\text{result}) \\
\text{VSR}[32\times T + T].\text{dword}(1) & \leftarrow 0xUUUU_UUUU_UUUU_UUUU \\
\text{FPRF} & \leftarrow \text{ClassSP}(\text{result}) \\
\text{FR} & \leftarrow \text{inc_flag} \\
\text{FI} & \leftarrow \text{xx_flag}
\end{align*}
\]

end else do

\[
\begin{align*}
\text{FR} & \leftarrow 00b \\
\text{FI} & \leftarrow 00b
\end{align*}
\]

end

Let XT be the value \(32\times T + T\).
Let XB be the value \(32\times B + B\).

Let src be the double-precision floating-point value in doubleword element 0 of VSR[XB].

src is rounded to single-precision using the rounding mode specified by RN.

See Table 50, “Scalar Floating-Point Intermediate Result Handling,” on page 515.

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 undefined.

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, VSR[XT] and FPRF are not modified.

Special Registers Altered

FPRF FR FI FX OX UX XX VXSNAN
**VSX Scalar Reciprocal Square Root Estimate Double-Precision XX2-form**

\[ \text{xsrsqrtedp} \rightarrow \text{XT, XB} \]

<table>
<thead>
<tr>
<th>60</th>
<th>6</th>
<th>T</th>
<th>//</th>
<th>16</th>
<th>B</th>
<th>21</th>
<th>74</th>
<th>B(7)</th>
</tr>
</thead>
</table>

\[
\begin{align*}
\text{XT} & \leftarrow \text{TX} \| \text{T} \\
\text{XB} & \leftarrow \text{BX} \| \text{B} \\
\text{reset}_\text{xflags}() \\
\text{v[0:inf]} & \leftarrow \text{ReciprocalSquareRootEstimateDP(VSR[XB][0:63])} \\
\text{result[0:63]} & \leftarrow \text{RoundToDP(RN, v)} \\
\text{if(vsnan_flag) then SetFX(VXSNAN)} \\
\text{if(vxsqrt_flag) then SetFX(VXSQRT)} \\
\text{if(zx_flag) then SetFX(ZX)} \\
\text{vex_flag} & \leftarrow \text{VE \& (vsnan_flag \| vxsqrt_flag)} \\
\text{zex_flag} & \leftarrow \text{ZE \& zx_flag} \\
\text{if(-vex_flag \& -zex_flag) then do} \\
& \text{VSR[XT]} \leftarrow \text{result[0:64]} \| 0xUUUU_UUUU_UUUU_UUUU \\
& \text{FPRF} \leftarrow \text{ClassDP(result)} \\
& \text{FR} \leftarrow \text{ObU} \\
& \text{FI} \leftarrow \text{ObU} \\
& \text{end} \\
\end{align*}

Let \( \text{XT} \) be the value \( 32 \times \text{TX} + \text{T} \).
Let \( \text{XB} \) be the value \( 32 \times \text{BX} + \text{B} \).

Let \( \text{src} \) 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 \( \text{src} \) is placed into doubleword element 0 of VSR[XT] in double-precision format.

Unless the reciprocal of the square root of \( \text{src} \) 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 \( \text{src} \). That is,

\[
\left| \frac{\text{estimate} - \frac{1}{\sqrt{\text{src}}}}{\frac{1}{\sqrt{\text{src}}} - \text{src}} \right| \leq \frac{1}{16384}
\]

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>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>SNaN</td>
<td>QNaN(^1)</td>
<td>VXSQAN</td>
</tr>
<tr>
<td>QNaN</td>
<td>QNaN(^1)</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 undefined.

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

- FPRF: FR=ObU FI=ObU FX
- XX=ObU VXSNAN VXSQRT

### VSR Data Layout for xsrsqrtedp

<table>
<thead>
<tr>
<th>src</th>
<th>VSR[XB]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>DP</td>
</tr>
<tr>
<td>tgt</td>
<td>VSR[XT]</td>
</tr>
<tr>
<td></td>
<td>DP</td>
</tr>
</tbody>
</table>

---

---
VSX Scalar Reciprocal Square Root Estimate
Single-Precision XX2-form

xsrsqtesp XT, XB

reset_xflags()

src ← VSR[32×XB].dword[0]
v ← ReciprocalSquareRootEstimateDP(src)
result ← RoundToSP(RN, v)

if(vxsnan_flag) then SetFX(VXSNAN)
if(vxsqrt_flag) then SetFX(VXSQRT)
if(ox_flag) then SetFX(OX)
if(ux_flag) then SetFX(UX)
if(0bU) then SetFX(XX)
if(zx_flag) then SetFX(ZX)

vex_flag ← VE & (vxsnan_flag | vxsqrt_flag)
zex_flag ← ZE & zx_flag

if( ~vex_flag & ~zex_flag ) then do
VSR[32×TX+T].dword[0] ← ConvertSPtoSP64(result)
VSR[32×TX+T].dword[1] ← 0xUUUU_UUUU_UUUU_UUUU
FPRF ← ClassSP(result)
FR ← 0bU
FI ← 0bU
end
else do
FR ← 0b0
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].

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,

\[ \frac{\text{estimate} - \frac{1}{\sqrt{src}}}{\frac{1}{\sqrt{src}}} \leq \frac{1}{16384} \]

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>–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>SNaN</td>
<td>QNaN(^1)</td>
<td>VXSQAN</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
undefined.

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 VXSQAN VXSQRT

VSR Data Layout for xsrsqtesp

src = VSR[XB]

dp = unused

tgt = VSR[XT]

dp = undefined
VSX Scalar Square Root Double-Precision XX2-form

xssqrtdp XT,XB

<table>
<thead>
<tr>
<th>60</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>75</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>XT</td>
<td>← TX</td>
<td></td>
<td>T</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XB</td>
<td>← BX</td>
<td></td>
<td>B</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

reset_xflags();

v[0:inf] ← SquareRootFP(VSR[XB][0:63])
result[0:63] ← RoundToDP(RN,v)
if(vxsnan_flag) then SetFX(VXSNAN)
if(vxsqrt_flag) then SetFX(VXSQRT)
if(xx_flag) then SetFX(XX)
ve_flag ← VE & (vxsnan_flag | vxsqrt_flag)
if(~ve_flag) then do
VSR[XT] ← result || 0xUUUU_UUUU_UUUU_UUUU
FPRF ← ClassDP(result)
FR ← inc_flag
FI ← xx_flag
end
else do
FR ← 0b0
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].
The unbounded-precision square root of src is produced.

The intermediate result is rounded to double-precision using the rounding mode specified by RN.

See Table 50, “Scalar Floating-Point Intermediate Result Handling,” on page 515.

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 undefined.

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 51, “VSX Scalar Floating-Point Final Result,” on page 516.

Special Registers Altered

FPRF FR FI FX XX VXSNAN VXSQRT

VSR Data Layout for xssqrtdp

src = VSR[XB]

tgt = VSR[XT]

<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>
</tbody>
</table>

vxsqrt_flag ← 1
vxsnan_flag ← 1

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.

Table 91.Actions for xssqrtdp
VSX Scalar Square Root Quad-Precision [using round to Odd] X-form

\[
\text{xssqrtqp} \quad \text{VRT,VRB} \quad (\text{RO}=0)
\]
\[
\text{xssqrtqpo} \quad \text{VRT,VRB} \quad (\text{RO}=1)
\]

Let \( s r c \) be the floating-point value in \( \text{VSR}[\text{VRB+32}] \) represented in quad-precision format.

If \( s r c \) is a Signalling NaN, an Invalid Operation exception occurs and \( \text{VXSQRT} \) is set to 1.

If \( s r c \) is a negative, non-zero value, an Invalid Operation exception occurs and \( \text{VXSQRT} \) is set to 1.

If \( s r c \) is a Signalling NaN, the result is the Quiet NaN corresponding to \( s r c \).

Otherwise, if \( s r c \) is a Quiet NaN, the result is \( s r c \).

Otherwise, if \( s r c \) is a negative value, the result is the default Quiet NaN[1].

Otherwise, do the following. The normalized square root of \( s r c \) is produced with unbounded significand precision and exponent range.

See Table 92, “Actions for xssqrtqp[o],” on page 643.

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 381 for a description of rounding modes.

If there is loss of precision, an Inexact exception occurs.

See Table 50, “Scalar Floating-Point Intermediate Result Handling,” on page 515.

The result is placed into \( \text{VSR}[\text{VRT+32}] \) in quad-precision format.

\( \text{FPFR} \) 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, \( \text{FPFR} \) 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 \( \text{FPFR} \) are not modified, and \( \text{FR} \) and \( \text{FI} \) are set to 0.

See Table 51, “VSX Scalar Floating-Point Final Result,” on page 516.

Special Registers Altered:

\( \text{FPFR} \ \text{FR} \ \text{FI} \ \text{FX} \ \text{VXSQRT} \ \text{XX} \)

### VSR Data Layout for xssqrtqp[o]

<table>
<thead>
<tr>
<th>VSR[VRT+32]</th>
<th>src</th>
</tr>
</thead>
<tbody>
<tr>
<td>VSR[VRT+32]</td>
<td>tgt</td>
</tr>
</tbody>
</table>
Chapter 7. Vector-Scalar Floating-Point Operations

Table 92. Actions for xssqrtqp[o]

<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 ← quiet(src)</td>
<td>xssqrt_flag ← 1</td>
</tr>
</tbody>
</table>

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.

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 XX2-form

xssqrtsp XT, XB

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].

The unbounded-precision square root of src is produced.

The intermediate result is rounded to single-precision using the rounding mode specified by RN.

See Table 50, “Scalar Floating-Point Intermediate Result Handling,” on page 515.

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 undefined.

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 51, “VSX Scalar Floating-Point Final Result,” on page 516.

Special Registers Altered

<table>
<thead>
<tr>
<th>FPRF</th>
<th>FR</th>
<th>FI</th>
<th>FX</th>
<th>OX</th>
<th>UX</th>
<th>XX</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

VXSNAN VXSQRT

VSR Data Layout for xssqrtsp

<table>
<thead>
<tr>
<th>src = VSR[XB]</th>
</tr>
</thead>
<tbody>
<tr>
<td>unused</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>tgt = VSR[XT]</th>
</tr>
</thead>
<tbody>
<tr>
<td>undefined</td>
</tr>
</tbody>
</table>

Table 93. Actions for xssqrtsp

<table>
<thead>
<tr>
<th>src</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Infinity</td>
</tr>
<tr>
<td>-NZF</td>
</tr>
<tr>
<td>-Zero</td>
</tr>
<tr>
<td>+Zero</td>
</tr>
<tr>
<td>+NZF</td>
</tr>
<tr>
<td>+Infinity</td>
</tr>
<tr>
<td>QNaN</td>
</tr>
<tr>
<td>SNaN</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.

Table 93. Actions for xssqrtsp
### VSX Scalar Subtract Double-Precision
#### XX3-form

#### xssubdp XT,XA,XB

<table>
<thead>
<tr>
<th>0</th>
<th>60</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>40</th>
<th>55</th>
</tr>
</thead>
<tbody>
<tr>
<td>XT</td>
<td>← TX</td>
<td></td>
<td>T</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XA</td>
<td>← AX</td>
<td></td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XB</td>
<td>← BX</td>
<td></td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>reset_xflags()</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>src1 ← VSR[XA]{0:63}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>src2 ← VSR[XB]{0:63}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>v[0:inf] ← AddDP(src1, NegateDP(src2))</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>result[0:63] ← RoundToDP(AN,v)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>if (vxsnan_flag) then SetFX(VXSNAN)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>if (vxisi_flag) then SetFX(VXISI)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>if (ox_flag) then SetFX(OX)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>if (ux_flag) then SetFX(UX)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>if (xx_flag) then SetFX(XX)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vex_flag ← VE &amp; (vxsnan_flag</td>
<td>vxisi_flag)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

if (~vex_flag) then do
  VSR[XT] ← result || 0xUUUU_UUUU_UUUU_UUUU |
  FPRF ← ClassDP(result) |
  FR ← inc_flag |
  FI ← xx_flag |
end else do
  FR ← 0b0 |
  FI ← 0b0 |
end

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[XT].

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 a sum having unbounded range and precision.

See Table 94.

The sum is normalized\(^2\).

The intermediate result is rounded to double-precision using the rounding mode specified by RN.

See Table 50, “Scalar Floating-Point Intermediate Result Handling,” on page 515.

---

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 94. Actions for xssubdp

<table>
<thead>
<tr>
<th>src2</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>(v \leftarrow \text{dQNaN})</td>
<td>(v \leftarrow -\infty)</td>
<td>(v \leftarrow -\infty)</td>
<td>(v \leftarrow -\infty)</td>
<td>(v \leftarrow -\infty)</td>
<td>(v \leftarrow \text{src2})</td>
<td>(\text{vxsnan_flag} \leftarrow 1)</td>
<td></td>
</tr>
<tr>
<td>(-\text{NZF})</td>
<td>(v \leftarrow +\text{NZF})</td>
<td>(v \leftarrow \text{src1})</td>
<td>(v \leftarrow \text{src1})</td>
<td>(v \leftarrow \text{src1})</td>
<td>(v \leftarrow -\text{src2})</td>
<td>(v \leftarrow -\text{src2})</td>
<td>(\text{vxsnan_flag} \leftarrow 1)</td>
<td></td>
</tr>
<tr>
<td>(-\text{Zero})</td>
<td>(v \leftarrow +\text{Zero})</td>
<td>(v \leftarrow -\text{src2})</td>
<td>(v \leftarrow -\text{Zero})</td>
<td>(v \leftarrow \text{Rezd})</td>
<td>(v \leftarrow -\text{src2})</td>
<td>(v \leftarrow -\text{src2})</td>
<td>(\text{vxsnan_flag} \leftarrow 1)</td>
<td></td>
</tr>
<tr>
<td>(+\text{Zero})</td>
<td>(v \leftarrow +\text{Zero})</td>
<td>(v \leftarrow -\text{src2})</td>
<td>(v \leftarrow \text{Rezd})</td>
<td>(v \leftarrow +\text{Zero})</td>
<td>(v \leftarrow -\text{src2})</td>
<td>(v \leftarrow -\text{src2})</td>
<td>(\text{vxsnan_flag} \leftarrow 1)</td>
<td></td>
</tr>
<tr>
<td>(+\text{NZF})</td>
<td>(v \leftarrow +\text{NZF})</td>
<td>(v \leftarrow \text{src1})</td>
<td>(v \leftarrow \text{src1})</td>
<td>(v \leftarrow \text{src1})</td>
<td>(v \leftarrow -\text{src2})</td>
<td>(v \leftarrow -\text{src2})</td>
<td>(\text{vxsnan_flag} \leftarrow 1)</td>
<td></td>
</tr>
<tr>
<td>(+\infty)</td>
<td>(v \leftarrow +\infty)</td>
<td>(v \leftarrow \text{src1})</td>
<td>(v \leftarrow \text{src1})</td>
<td>(v \leftarrow \text{src1})</td>
<td>(v \leftarrow -\text{src2})</td>
<td>(v \leftarrow -\text{src2})</td>
<td>(\text{vxsnan_flag} \leftarrow 1)</td>
<td></td>
</tr>
<tr>
<td>QNaN</td>
<td>(v \leftarrow \text{src1})</td>
<td>(v \leftarrow \text{src1})</td>
<td>(v \leftarrow \text{src1})</td>
<td>(v \leftarrow \text{src1})</td>
<td>(v \leftarrow -\text{src2})</td>
<td>(v \leftarrow -\text{src2})</td>
<td>(\text{vxsnan_flag} \leftarrow 1)</td>
<td></td>
</tr>
<tr>
<td>SNaN</td>
<td>(v \leftarrow \text{Q(src1)})</td>
<td>(\text{vxsnan_flag} \leftarrow 1)</td>
<td>(v \leftarrow \text{Q(src1)})</td>
<td>(\text{vxsnan_flag} \leftarrow 1)</td>
<td>(v \leftarrow \text{Q(src1)})</td>
<td>(\text{vxsnan_flag} \leftarrow 1)</td>
<td>(\text{vxsnan_flag} \leftarrow 1)</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[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): 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.
Chapter 7. Vector-Scalar Floating-Point Operations

VSX Scalar Subtract Quad-Precision [using round to Odd] X-form

xssubqp VRT, VRA, VRB (RO=0)
xssubqpo VRT, VRA, VRB (RO=1)

<table>
<thead>
<tr>
<th>63</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>516</th>
<th>21</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>VRT</td>
<td>VRA</td>
<td>VRB</td>
<td>VRT+32</td>
<td></td>
</tr>
</tbody>
</table>

Let \( src_1 \) be the floating-point value in \( VSR[VRA+32] \) represented in quad-precision format.

Let \( src_2 \) be the floating-point value in \( VSR[VRB+32] \) represented in quad-precision format.

If either \( src_1 \) or \( src_2 \) is a Signalling NaN, an Invalid Operation exception occurs and \( VXSNAN \) is set to 1.

If \( src_1 \) and \( src_2 \) are Infinity values having same signs, an Invalid Operation exception occurs and \( VXISI \) is set to 1.

If \( src_1 \) is a Signalling NaN, the result is the Quiet NaN corresponding to \( src_1 \).

Otherwise, if \( src_1 \) is a Quiet NaN, the result is \( src_1 \).

Otherwise, if \( src_2 \) is a Signalling NaN, the result is the Quiet NaN corresponding to \( src_2 \).

Otherwise, if \( src_2 \) is a Quiet NaN, the result is \( src_2 \).

Otherwise, if \( src_1 \) and \( src_2 \) are Infinity values having same signs, the result is the default Quiet NaN\(^1\).

1. The quad-precision default Quiet NaN is the value, \( 0x7FFFFFFF_0000_0000_0000_0000_0000_0000 \).

Otherwise, do the following.

The normalized sum of the negation of \( src_2 \) added to \( src_1 \) is produced with unbounded significand precision and exponent range.

See Table 95, “Actions for xssubqp[o],” on page 648.

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.

See Table 50, “Scalar Floating-Point Intermediate Result Handling,” on page 515.

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.

If a trap-disabled Invalid Operation exception occurs, \( FPREF \) 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 \( FPREF \) are not modified, and \( FR \) and \( FI \) are set to 0.

See Table 51, “VSX Scalar Floating-Point Final Result,” on page 516.

Special Registers Altered:

\[
\begin{align*}
\text{FPREF} & \quad \text{FR} & \quad \text{FI} & \quad \text{FX} & \quad \text{VXSNAN} & \quad \text{VXISI} & \quad \text{OX} & \quad \text{UX} & \quad \text{XX}
\end{align*}
\]

VSR Data Layout for xssubqp[o]

\( VSR[VRA+32] \)

\( VSR[VRB+32] \)

\( VSR[VRT+32] \)

\( tgt \)
### Table 95. 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><strong>src2</strong></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>v + 0NaN</td>
<td></td>
<td></td>
<td></td>
<td>v + 0NaN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-NZF</td>
<td>v + sub(src1,src2)</td>
<td>v + src1</td>
<td></td>
<td></td>
<td></td>
<td>v + sub(src1,src2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-Zero</td>
<td>v + src1</td>
<td>v + Rez</td>
<td>v + 0NaN</td>
<td></td>
<td>v + src1</td>
<td>v + src1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+Zero</td>
<td>v + Rez</td>
<td>v + 0NaN</td>
<td>v + src1</td>
<td></td>
<td>v + 0NaN</td>
<td>v + Rez</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+NZF</td>
<td>v + sub(src1,src2)</td>
<td>v + src1</td>
<td></td>
<td></td>
<td></td>
<td>v + sub(src1,src2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+Infinity</td>
<td>v + 0NaN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v + 0NaN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>QNaN</td>
<td>v + src1</td>
<td>v + quiet(src1)</td>
<td></td>
<td></td>
<td></td>
<td>v + src1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SNaN</td>
<td>v + quiet(src1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v + quiet(src1)</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].
- **0NaN** 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

Reset_xflags();

src1 ← VSR[32×AX+A].dword[0];
src2 ← VSR[32×BX+B].dword[0];
v ← AddDP(src1, NegateDP(src2));
result ← RoundToSP(RN, v);

if(vxsnan_flag) then SetFX(VXSNAN);
if(vxisi_flag) then SetFX(VXISI);
if(ox_flag) then SetFX(OX);
if(ux_flag) then SetFX(UX);
if(xx_flag) then SetFX(XX);

vex_flag ← VE & (vxsnan_flag | vxisi_flag);

if(~vex_flag) then

VSR[32×TX+T].dword[0] ← ConvertSPtoSP64(result);
VSR[32×TX+T].dword[1] ← 0xUUUU_UUUU_UUUU_UUUU;
FPRF ← ClassSP(result);
FR ← inc_flag;
FI ← xx_flag;
else do
FR ← 0b0;
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].

src2 is negated and added\(^1\) to src1, producing the sum, v, having unbounded range and precision.

See Table 96, “Actions for xssubsp,” on page 650.

v is normalized\(^2\) and rounded to single-precision using the rounding mode specified by RN.

See Table 50, “Scalar Floating-Point Intermediate Result Handling,” on page 515.

---

\(^1\) 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)**: 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 96. 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 ← 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>vxsnan_flag ← 1</td>
</tr>
<tr>
<td>-Zero</td>
<td>v ← +Infinity</td>
<td>v ← -src2</td>
<td>v ← -src2</td>
<td>v ← -src2</td>
<td>v ← -src2</td>
<td>v ← -Infinity</td>
<td>v ← 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>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>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 ← +Infinity</td>
<td>v ← 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 Scalar Test for software Divide Double-Precision XX3-form

xstdivdp BF,XA,XB

<table>
<thead>
<tr>
<th></th>
<th>60</th>
<th>61</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>BF</td>
<td>//</td>
</tr>
<tr>
<td>A</td>
<td>9</td>
<td>16</td>
</tr>
<tr>
<td>B</td>
<td>21</td>
<td></td>
</tr>
</tbody>
</table>

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].

Let src2 be the double-precision floating-point value in doubleword element 0 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.

Otherwise fe_flag is set to 0.

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.

Otherwise fg_flag is set to 0.

CR[BF] is set to the value $0b1 || fg_flag || fe_flag || 0b0$.

Special Registers Altered
**VSX Scalar Test for software Square Root**

**Double-Precision XX2-form**

xstsqrtdp BF,XB

<table>
<thead>
<tr>
<th>0</th>
<th>60</th>
<th>BF</th>
<th>//</th>
<th>//</th>
<th>B</th>
<th>106</th>
<th>BX</th>
</tr>
</thead>
</table>

18 \xrightarrow{=} 32 \times BX + B

cr \xleftarrow{=} VSR[XB][0:63]

e_b \xleftarrow{=} VSR[XB][1:11] - 1023

fe_flag \xleftarrow{=} \text{IsNaN(src)} | \text{IsInf(src)} | \text{IsZero(src)} | \text{IsNeg(src)} | (e_b \leq -970)

fg_flag \xleftarrow{=} \text{IsInf(src)} | \text{IsZero(src)} | \text{IsDen(src)}

fl_flag \xleftarrow{=} \text{xsrsqrtedp_error}() \leq 2^{-14}

CR[BF] \xleftarrow{=} 0b1 || fg_flag || fe_flag || 0b0

Let X8 be the value $32 \times BX + B$.

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.

fg_flag is set to 1 for any of the following conditions.
- src is a zero, an infinity, or a denormalized value.

Otherwise fg_flag is set to 0.

CR field BF is set to the value 0b1 || fg_flag || fe_flag || 0b0.

**Special Registers Altered**

CR[BF]

**VSR Data Layout for xstsqrtdp**

src = VSR[XB]

<table>
<thead>
<tr>
<th>0</th>
<th>64</th>
<th>DP</th>
<th>unused</th>
<th>127</th>
</tr>
</thead>
</table>
Chapter 7. Vector-Scalar Floating-Point Operations

VSX Scalar Test Data Class Double-Precision XX2-form

\[ \text{VSX Scalar Test Data Class Double-Precision XX2-form} \]
\[ \text{xststdcdp BF,XB,DCMX} \]

<table>
<thead>
<tr>
<th>( \text{BF} )</th>
<th>( \text{XB} )</th>
<th>( \text{DCMX} )</th>
<th>( \text{B} )</th>
<th>( \text{362} )</th>
<th>( \text{bit} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>60</td>
<td>6</td>
<td>9</td>
<td>0</td>
<td>p1</td>
<td>362</td>
</tr>
</tbody>
</table>

if MSR.VSX=0 then VSX.Unavailable

\[ \text{src} \leftarrow \text{VSR[32×BX+B].dword}[0] \]
\[ \text{exponent} \leftarrow \text{src.bit}[1:11] \]
\[ \text{fraction} \leftarrow \text{src.bit}[12:63] \]
\[ \text{class.Infinity} \leftarrow (\text{exponent} = 0x7FF) \& (\text{fraction} = 0) \]
\[ \text{class.NaN} \leftarrow (\text{exponent} = 0x7FF) \& (\text{fraction} \neq 0) \]
\[ \text{class.Zero} \leftarrow (\text{exponent} = 0x000) \& (\text{fraction} = 0) \]
\[ \text{class.Denormal} \leftarrow (\text{exponent} = 0x000) \& (\text{fraction} \neq 0) \]

\[ \text{if MSR.VSX=0 then VSX.Unavailable} \]

\[ \text{src} \leftarrow \text{VSR[32×BX+B].dword}[0] \]
\[ \text{exponent} \leftarrow \text{src.bit}[1:11] \]
\[ \text{fraction} \leftarrow \text{src.bit}[12:63] \]
\[ \text{class.Infinity} \leftarrow (\text{exponent} = 0x7FF) \& (\text{fraction} = 0) \]
\[ \text{class.NaN} \leftarrow (\text{exponent} = 0x7FF) \& (\text{fraction} \neq 0) \]
\[ \text{class.Zero} \leftarrow (\text{exponent} = 0x000) \& (\text{fraction} = 0) \]
\[ \text{class.Denormal} \leftarrow (\text{exponent} = 0x000) \& (\text{fraction} \neq 0) \]

match
\[ \text{DCMX.bit}[0] \& \text{class.NaN} \]
\[ \text{DCMX.bit}[1] \& \text{class.Infinity} \& \text{!sign} \]
\[ \text{DCMX.bit}[2] \& \text{class.Infinity} \& \text{sign} \]
\[ \text{DCMX.bit}[3] \& \text{class.Zero} \& \text{!sign} \]
\[ \text{DCMX.bit}[4] \& \text{class.Zero} \& \text{sign} \]
\[ \text{DCMX.bit}[5] \& \text{class.Denormal} \& \text{!sign} \]
\[ \text{DCMX.bit}[6] \& \text{class.Denormal} \& \text{sign} \]

\[ \text{CR.bit}[4×BF] \leftarrow \text{FPSCR.FL} \leftarrow \text{src.sign} \]
\[ \text{CR.bit}[4×BF+1] \leftarrow \text{FPSCR.FG} \leftarrow 0b0 \]
\[ \text{CR.bit}[4×BF+2] \leftarrow \text{FPSCR.FE} \leftarrow \text{match} \]
\[ \text{CR.bit}[4×BF+3] \leftarrow \text{FPSCR.FU} \leftarrow 0b0 \]

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>64</td>
<td>127</td>
</tr>
</tbody>
</table>

Let \( \text{XB} \) be the sum \( 32\times\text{BX} + \text{B} \).

Let \( \text{src} \) be the double-precision floating-point value in doubleword element 0 of \( \text{VSR[XB]} \).

Bit 0 of \( \text{CR} \) field \( \text{BF} \) and bit 0 of \( \text{FPCC} \) are set to the sign bit of \( \text{src} \).

Bit 1 of \( \text{CR} \) field \( \text{BF} \) and bit 1 of \( \text{FPCC} \) are set to 0b0.

Bit 2 of \( \text{CR} \) field \( \text{BF} \) and bit 2 of \( \text{FPCC} \) are set to indicate whether the data class of \( \text{src} \), as represented in double-precision format, matches any of the data classes specified by \( \text{DCMX} \) (Data Class Mask).

DCMX bit Data Class
0 NaN
1 +Infinity
2 -Infinity
3 +Zero
4 -Zero
5 +Denormal
6 -Denormal

Bit 3 of \( \text{CR} \) field \( \text{BF} \) and bit 3 of \( \text{FPCC} \) are set to 0b0.

Special Registers Altered:
\( \text{CR} \) field BF
\( \text{FPCC} \)
Let $src$ be the quad-precision floating-point value in $VSR[VRB+32]$.

Let the 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>$+\infty$</td>
</tr>
<tr>
<td>2</td>
<td>$-\infty$</td>
</tr>
<tr>
<td>3</td>
<td>$+0$</td>
</tr>
<tr>
<td>4</td>
<td>$-0$</td>
</tr>
<tr>
<td>5</td>
<td>Denormal</td>
</tr>
<tr>
<td>6</td>
<td>Denormal</td>
</tr>
</tbody>
</table>

Bit 0 of CR field BF and bit 0 of FPCC are set to the sign 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 quad-precision format, matches any of the data classes specified by DCM.

Bit 3 of CR field BF and bit 3 of FPCC are set to 0b0.

**Special Registers Altered:**
- CR field BF
- FPCC
VSX Scalar Test Data Class Single-Precision XX2-form

**xtstdcsp BF,XB,DCMX**

<table>
<thead>
<tr>
<th>60</th>
<th>BF</th>
<th>DCMX</th>
<th>B</th>
<th>p1</th>
<th>298</th>
<th>Bf/p1</th>
</tr>
</thead>
</table>

- Let $XB$ be the sum $32 \times BX + B$.
- Let $src$ be the double-precision floating-point value in doubleword element 0 of $VSR[XB]$.

**Special Registers Altered:**

- CR field BF
- CR field FG
- CR field FE
- CR field FU
- VSR Data Layout for xtstdcdp

<table>
<thead>
<tr>
<th>src</th>
<th>VSR[XB].dword[0]</th>
<th>unused</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>64</td>
<td>127</td>
</tr>
</tbody>
</table>

**if MSR.VSX=0 then VSX_Unavailable();**

```c
src ← VSR[32×BX+BX].dword[0]
exponent ← src.bit[1:11]
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)
not_SP_value ← (src != Convert_SPtoDP(Convert_DPtoSP(src)))
```
VSX Scalar Extract Exponent Double-Precision XX2-form

**xsxexpdp RT,XB**

<table>
<thead>
<tr>
<th>60</th>
<th>RT</th>
<th>0</th>
<th>B</th>
<th>347</th>
<th>BI</th>
</tr>
</thead>
</table>

Let $XB$ be the sum $32 \times BX + B$.

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

**xsxexpqp VRT,VRB**

<table>
<thead>
<tr>
<th>63</th>
<th>VRT</th>
<th>2</th>
<th>VRB</th>
<th>804</th>
<th>BI</th>
</tr>
</thead>
</table>

if MSR.VSX=0 then VSX_Unavailable()

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**

<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>

08 63 127

**VSR Data Layout for xsxexpdp**

<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>

08 63 127
Chapter 7. Vector-Scalar Floating-Point Operations

### VSX Scalar Extract Significand

#### Double-Precision XX2-form

**xsxsigdp**

<table>
<thead>
<tr>
<th>60</th>
<th>RT</th>
<th>1</th>
<th>16</th>
<th>B</th>
<th>347</th>
<th>0</th>
<th>31</th>
</tr>
</thead>
</table>

- if MSR.VSX=0 then VSX_Unavailable()

- exponent ← VSR[32×BX+B].bit[1:12]
- fraction ← EXTZ64(VSR[32×BX+B].bit[12:63])

- if (exponent != 0) & (exponent != 2047) then
  - significand ← fraction | 0x0000_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

---

#### Quad-Precision X-form

**xsxsigqp**

<table>
<thead>
<tr>
<th>63</th>
<th>VRT</th>
<th>18</th>
<th>VRB</th>
<th>804</th>
<th>0</th>
<th>31</th>
</tr>
</thead>
</table>

- if MSR.VSX=0 then VSX_Unavailable()

- src ← VSR[VRB+32]
- exponent ← EXTZ(src.bit[1:15])
- fraction ← EXTZ128(src.bit[16:237])

- if (exponent != 0) & (exponent != 32767) then
  - VSR[VRT+32] ← fraction | 0x0000_0000_0000_0000_0000_0000_0000_0000
  - else
  - VSR[VRT+32] ← fraction

Let **src** be the quad-precision floating-point value in VSR[VRB+32].

The significand of **src** is placed into VSR[VRT+32].

If the value of the exponent field of **src** is equal to 0b000_0000_0000_0000 (i.e., Zero or Denormal value) or 0b111_1111_1111 (i.e., Infinity or NaN), 0b0 is placed into bit 15 of VSR[VRT+32]. Otherwise (i.e., Normal value), 0b1 is placed into bit 15 of VSR[VRT+32]. The contents of bits 0:14 of VSR[VRT+32] are set to 0.

**Special Registers Altered:**

None

---

#### 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>

<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>

---

**Programming Note**

This instruction can be used to operate on a single-precision source operand.
**VSX Vector Absolute Value Double-Precision XX2-form**

\[
xvabsdp \quad XT, XB
\]

|       | 60 | T | || | B | 473 | DT |
|-------|----|---|---|---|---|-----|----|
| XT    | TX || T |
| XB    | BX || B |

do i=0 to 127 by 64
  VSR[XT]{i:i+63} ← 0b0 || VSR[XB]{i+1:i+63}
end

Let XT be the value \(32 \times TX + T\).

Let XB be the value \(32 \times BX + B\).

For each vector element 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**

<table>
<thead>
<tr>
<th>src = VSR[XB]</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>DP</td>
<td>DP</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>tgt = VSR[XT]</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>DP</td>
<td>DP</td>
</tr>
</tbody>
</table>

**VSX Vector Absolute Value Single-Precision XX2-form**

\[
xvabssp \quad XT, XB
\]

|       | 60 | T | || | B | 409 | DT |
|-------|----|---|---|---|---|-----|----|
| XT    | TX || T |
| XB    | BX || B |

do i=0 to 127 by 32
  VSR[XT]{i:i+31} ← 0b0 || VSR[XB]{i+1:i+31}
end

Let XT be the value \(32 \times TX + T\).

Let XB be the value \(32 \times BX + B\).

For each vector element 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**

<table>
<thead>
<tr>
<th>src = VSR[XB]</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>SP</td>
<td>SP</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>tgt = VSR[XT]</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>SP</td>
<td>SP</td>
</tr>
</tbody>
</table>

0 | 64 | 127
Chapter 7. Vector-Scalar Floating-Point Operations

VSX Vector Add Double-Precision XX3-form

The result is placed into doubleword element i of VSR[XT] in double-precision format.

See Table 98, "Vector Floating-Point Final Result," on page 661.

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 |

VSR Data Layout for xvadddp

For each vector element 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].

src2 is added\(^1\) to src1, producing a sum having unbounded range and precision.

The sum is normalized\(^2\).

See Table 97.

The intermediate result is rounded to double-precision using the rounding mode specified by RN.

See Table 50, "Scalar Floating-Point Intermediate Result Handling," on page 515.

---

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 97. Actions for xvadddp (element i)

<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 ← dQNaN</td>
<td>vxsi_flag ← 1</td>
<td>v ← src1</td>
<td>v ← -Q(src1)</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>
</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 ← 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>
</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 ← 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>vxsi_flag ← 1</td>
<td>v ← -Q(src1)</td>
<td>vxsi_flag ← 1</td>
<td>v ← -Q(src1)</td>
<td>vxsi_flag ← 1</td>
<td>v ← -Q(src1)</td>
<td>vxsi_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).
- **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.
## Case

<table>
<thead>
<tr>
<th>Case</th>
<th>VE</th>
<th>OE</th>
<th>UE</th>
<th>ZE</th>
<th>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>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Special</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Special</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Special</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Special</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Special</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Special</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Special</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Special</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Normal</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>-</td>
<td>-</td>
</tr>
<tr>
<td>Normal</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>yes</td>
<td>no</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Normal</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>yes</td>
<td>yes</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Normal</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>no</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Normal</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>yes</td>
<td>yes</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

### Explanation:
- The results do not depend on this condition.
- $fx(x)$: FX is set to 1 if $x=0$. x is set to 1.
- The value defined in Table 50, “Scalar Floating-Point Intermediate Result Handling,” on page 515, significand rounded to the target precision, unbounded exponent range.
- The value defined in Table 50, “Scalar Floating-Point Intermediate Result Handling,” on page 515, significand rounded to the target precision, bounded exponent range.
- The precise intermediate result defined in the instruction having unbounded significand precision, unbounded exponent range.
- OX: Floating-Point Overflow Exception status flag, FPSCR_{OX}.
- error(): The system error handler is invoked for the trap-enabled exception if the FE0 and FE1 bits in the Machine State Register are set to any mode other than the ignore-exception mode. Update of the target VSR is suppressed for all vector elements.
- The value $x$ is placed in element $i$ of VSR[XT] in the target precision format (where $i \in [0,1]$ for results with 64-bit elements, and $i \in [0,1,3,4]$ for results with 32-bit elements).
- UX: Floating-Point Underflow Exception status flag, FPSCR_{UX}.
- VXSQRT: Floating-Point Invalid Operation Exception (SNan) status flag, FPSCR_{VXSQRT}.
- VXIDI: Floating-Point Invalid Operation Exception (Infinity ÷ Infinity) status flag, FPSCR_{VXIDI}.
- VXIMZ: Floating-Point Invalid Operation Exception (Infinity × Zero) status flag, FPSCR_{VXIMZ}.
- VXISI: Floating-Point Inexact Exception status flag, FPSCR_{VXISI}.
- VXZDZ: Floating-Point Inexact Exception status flag, FPSCR_{VXZDZ}.
- XX: Floating-Point Inexact Exception status flag, FPSCR_{XX}.
- ZX: Floating-Point Zero Divide Exception status flag, FPSCR_{ZX}.

### Table 98. Vector Floating-Point Final Result
<table>
<thead>
<tr>
<th>Case</th>
<th>VE</th>
<th>OX</th>
<th>UE</th>
<th>ZE</th>
<th>XE</th>
<th>vximz_flag</th>
<th>vxidi_flag</th>
<th>vxzdz_flag</th>
<th>vxsimz_flag</th>
<th>vxsqrt_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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>T(1), fx(OX), fx(XX)</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>T(1), fx(OX), fx(XX), error()</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>fx(OX),</td>
<td>error()</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>yes no</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>yes yes</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>fx(OX), fx(XX), error()</td>
<td></td>
</tr>
<tr>
<td>Tiny</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>T(1)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>yes no</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>yes yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>yes yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>yes no</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>yes yes</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>fx(UX), fx(XX), error()</td>
<td></td>
</tr>
</tbody>
</table>

Explanation:
- The results do not depend on this condition.
- fx(x) FX is set to 1 if x=0, x is set to 1.
- q The value defined in Table 50, “Scalar Floating-Point Intermediate Result Handling,” on page 515, significand rounded to the target precision, unbounded exponent range.
- r The value defined in Table 50, “Scalar Floating-Point Intermediate Result Handling,” on page 515, 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.
- OX Floating-Point Overflow Exception status flag, FPSCR_OX.
- error() The system error handler is invoked for the trap-enabled exception if the FE0 and FE1 bits in the Machine State Register 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).
- UX Floating-Point Underflow Exception status flag, FPSCR_UX.
- VXSNAN Floating-Point Invalid Operation Exception (SNan) status flag, FPSCR_VXSNAN.
- VXSQRT Floating-Point Invalid Operation Exception (Invalid Square Root) status flag, FPSCR_VXSQRT.
- VXIDI Floating-Point Invalid Operation Exception (Infinity ÷ Infinity) status flag, FPSCR_VXIDI.
- VXIMZ Floating-Point Invalid Operation Exception (Infinity × Zero) status flag, FPSCR_VXIMZ.
- VXISI Floating-Point Invalid Operation Exception (Infinity – Infinity) status flag, FPSCR_VXISI.
- VXZDZ Floating-Point Invalid Operation Exception (Zero ÷ Zero) status flag, FPSCR_VXZDZ.
- XX Floating-Point Inexact Exception status flag, FPSCR_XX. The flag is a sticky version of FPSCR_R. When FPSCR_R is set to a new value, the new value of FPSCR_XX is set to the result of ORing the old value of FPSCR_XX with the new value of FPSCR_R.
- ZX Floating-Point Zero Divide Exception status flag, FPSCR_ZX.

Table 98.Vector Floating-Point Final Result (Continued)
VSX Vector Add Single-Precision XX3-form

xvaddsp XT,XA,XB

<table>
<thead>
<tr>
<th></th>
<th>60</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>26</th>
</tr>
</thead>
<tbody>
<tr>
<td>XT</td>
<td>← TX</td>
<td></td>
<td>T</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XA</td>
<td>← AX</td>
<td></td>
<td>A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XB</td>
<td>← BX</td>
<td></td>
<td>B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ex_flag</td>
<td>← 0b0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
\text{do i=0 to 127 by 32}
\]
\[
\text{reset_xflags}()
\]
\[
\text{src1} ← VSR[XA]\{i:i+31\}
\]
\[
\text{src2} ← VSR[XB]\{i:i+31\}
\]
\[
\text{result}\{i:i+31\} ← \text{AddP(src1, src2)}
\]
\[
\text{if(vxsnan_flag) then SetFX(VXSNAN)}
\]
\[
\text{if(vxisi_flag) then SetFX(VXISI)}
\]
\[
\text{if(ox_flag) then SetFX(OX)}
\]
\[
\text{if(ux_flag) then SetFX(UX)}
\]
\[
\text{if(xx_flag) then SetFX(XX)}
\]
\[
\text{ex_flag} ← \text{ex_flag} | (VE & vxsnan_flag)
\]
\[
\text{ex_flag} ← \text{ex_flag} | (VE & vxisi_flag)
\]
\[
\text{ex_flag} ← \text{ex_flag} | (OE & ox_flag)
\]
\[
\text{ex_flag} ← \text{ex_flag} | (UE & ux_flag)
\]
\[
\text{ex_flag} ← \text{ex_flag} | (XE & xx_flag)
\]
\[
\text{end}
\]
\[
\text{if( ex_flag = 0 ) then VSR[XT] ← result}
\]

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 vector element 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 added\(^1\) to `src1`, producing a sum having unbounded range and precision.

The sum is normalized\(^2\).

See Table 99.

The intermediate result is rounded to single-precision using the rounding mode specified by RN.

See Table 50, “Scalar Floating-Point Intermediate Result Handling,” on page 515.

---

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 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 99. Actions for `xvaddsp` (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>&quot;-Infinity&quot;</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>v ← src2</td>
<td>v ← Q(src2)</td>
</tr>
<tr>
<td>&quot;-NZF&quot;</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>
</tr>
<tr>
<td>&quot;-Zero&quot;</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>&quot;+Zero&quot;</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>&quot;+Infinity&quot;</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>&quot;QNaN&quot;</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>&quot;SNaN&quot;</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 Compare Equal To Double-Precision XX3-form

\[
\begin{align*}
\text{xvcmpeqdp} & : XT,XA,XB \ (Rc=0) \\
\text{xvcmpeqdp} & : XT,XA,XB \ (Rc=1)
\end{align*}
\]

<table>
<thead>
<tr>
<th>60</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>22</th>
<th>99</th>
<th>128</th>
</tr>
</thead>
<tbody>
<tr>
<td>XT</td>
<td>TX</td>
<td></td>
<td>T</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XA</td>
<td>AX</td>
<td></td>
<td>A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XB</td>
<td>BX</td>
<td></td>
<td>B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ex_flag</td>
<td>000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>all_false</td>
<td>001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>all_true</td>
<td>001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\begin{algorithm}
do i=0 to 127 by 64
    reset_xflags()
    src1 ← VSR[XA]_{i:i+63}
    src2 ← VSR[XB]_{i:i+63}
    vxsnan_flag ← IsSNaN(src1) | IsSNaN(src2)
    if( CompareEQDP(src1,src2) ) then
        result_{i:i+63} ← 0xFFFF_FFFF_FFFF_FFFF
        all_false ← 0b0
    end
    else do
        result_{i:i+63} ← 0x0000_0000_0000_0000
        all_true ← 0b0
    end
    if(vxsnan_flag) then SetFX(VXSNAN)
    ex_flag ← ex_flag | (VE & vxsnan_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result
if(Rc=1) then do
    if( !vex_flag ) then
        CR[6] ← all_true || 0b0 || all_false || 0b0
    else
        CR[6] ← 0UUUU
    end
end

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 vector element \(i\) 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 of CR[6] is set to indicate all vector elements compared true.
- Bit 1 of CR[6] is set to 0.
- Bit 2 of CR[6] is set to indicate all vector elements compared false.
- Bit 3 of CR[6] 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[6] are undefined if Rc is equal to 1.

Special Registers Altered

CR[6] ............... (if Rc=1)
FX : VXSNAN

VSR Data Layout for xvcmpeqdp[]

<table>
<thead>
<tr>
<th>src1 = VSR[XA]</th>
<th>src2 = VSR[XB]</th>
</tr>
</thead>
<tbody>
<tr>
<td>DP</td>
<td>DP</td>
</tr>
<tr>
<td>DP</td>
<td>DP</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>tgt = VSR[XT]</th>
</tr>
</thead>
<tbody>
<tr>
<td>MD</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0</th>
<th>64</th>
<th>127</th>
</tr>
</thead>
<tbody>
<tr>
<td>MD</td>
<td>0</td>
<td></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 \(32 \times BX + B\).
VSX Vector Compare Equal To Single-Precision XX3-form

\[ \text{xvcmpeqsp.} \quad \text{XT,XA,XB} \quad (\text{Rc=1}) \]

<table>
<thead>
<tr>
<th>0</th>
<th>60</th>
<th>6</th>
<th>1</th>
<th>A</th>
<th>B</th>
<th>Rd</th>
<th>67</th>
</tr>
</thead>
</table>

Let \( \text{XT} \) be the value \( 32 \times \text{TX} + \text{T} \).
Let \( \text{XA} \) be the value \( 32 \times \text{AX} + \text{A} \).
Let \( \text{XB} \) be the value \( 32 \times \text{BX} + \text{B} \).

For each vector element \( i \) from 0 to 3, do the following.

Let \( \text{src1} \) be the single-precision floating-point operand in word element \( i \) of VSR[XT].

Let \( \text{src2} \) be the single-precision floating-point operand in word element \( i \) of VSR[XB].

\( \text{src1} \) is compared to \( \text{src2} \).

The contents of word element \( i \) of VSR[XT] are set to all 1s if \( \text{src1} \) is equal to \( \text{src2} \), and is set to all 0s otherwise.

A NaN input causes the comparison to return false for that element.
**VSX Vector Compare Greater Than or Equal To Double-Precision XX3-form**

\[
\begin{align*}
\text{xvcmpgedp} & \quad \text{XT,XA,XB (Rc=0)} \\
\text{xvcmpgedp.} & \quad \text{XT,XA,XB (Rc=1)} \\
\end{align*}
\]

<table>
<thead>
<tr>
<th>60</th>
<th>6</th>
<th>T</th>
<th>A</th>
<th>B</th>
<th>Rc</th>
<th>115</th>
<th>VSR[XT]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>115</td>
<td></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 \( 32 \times BX + B\).

For each vector element \(i\) 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 \(src2\).
VSX Vector Compare Greater Than or Equal To Single-Precision XX3-form

For each vector element \( i \) 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].

\( \text{src1} \) is compared to \( \text{src2} \).
### VSX Vector Compare Greater Than Double-Precision XX3-form

```plaintext
xvcmpgtdp XT,XA,XB (Rc=0)
xvcmpgtdp XT,XA,XB (Rc=1)
```

<table>
<thead>
<tr>
<th>i</th>
<th>XT</th>
<th>T</th>
<th>A</th>
<th>B</th>
<th>Rc</th>
<th>107</th>
<th>127</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

- **XT** = TX || T
- **XA** = AX || A
- **XB** = BX || B
- **ex_flag** = 0b0
- **all_false** = 0b1
- **all_true** = 0b1

```plaintext
do i=0 to 127 by 64
  reset_xflags()
  src1 ← VSR[XA][i:i+63]
  src2 ← VSR[XB][i:i+63]
  if( IsSNaN(src1) | IsSNaN(src2) ) then do
    vxsnan_flag ← 0b1
    if(VE=0) then vxvc_flag ← 0b1
  end
  else vxvc_flag ← IsQNaN(src1) | IsQNaN(src2)
  if( CompareGTDP(src1,src2) ) then do
    result[i:i+63] ← 0xFFFF_FFFF_FFFF_FFFF
    all_false ← 0b0
  end
  else do
    result[i:i+63] ← 0x0000_0000_0000_0000
    all_true ← 0b0
  end
  if( vxsnan_flag ) then SetFX(VXSNAN)
  if( vxvc_flag ) then SetFX(VXVC)
  ex_flag ← ex_flag | (VE & vxsnan_flag)
  ex_flag ← ex_flag | (VE & vxvc_flag)
end
```

- **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 of CR[6] is set to indicate all vector elements compared true.
- Bit 1 of CR[6] is set to 0.
- Bit 2 of CR[6] is set to indicate all vector elements compared false.
- Bit 3 of CR[6] 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[6] are undefined if Rc is equal to 1.

### Special Registers Altered
- CR[6] .............................. (if Rc=1)
- FX  VXSNAN  VXVC

### VSR Data Layout for xvcmpgtdp[]

| src1 = VSR[XA]| src2 = VSR[XB] |
| DP | DP |
| DP | DP |

| tgt = VSR[XT]| |
| MD | MD | 127 |

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 vector element i from 0 to 1, do the following.
- **src1** is compared to **src2**.

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].
VSX Vector Compare Greater Than
Single-Precision XX3-form

xvcmpgtsp XT,XA,XB (Rc=0)

tabular:

<table>
<thead>
<tr>
<th>T</th>
<th>A</th>
<th>B</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>60</td>
<td>6</td>
<td>11</td>
<td>16</td>
</tr>
<tr>
<td>22</td>
<td>75</td>
<td>80</td>
<td>81</td>
</tr>
</tbody>
</table>

XT ← TX || T
XA ← AX || A
XB ← BX || B
ex_flag ← 0b0
all_false ← 0b1
all_true ← 0b1

\[
\begin{array}{c}
\text{do } i=0 \text{ to } 327 \text{ by } 32 \\
\text{ reset_allflags()} \\
\text{ src1 ← VSR[XA]{i:i+31} } \\
\text{ src2 ← VSR[XB]{i:i+31} } \\
\text{ if( IsSNaN(src1) | IsSNaN(src2) ) then do } \\
\text{ vxsnan_flag ← 0b1 } \\
\text{ if(VE=0) then vxvc_flag ← 0b1 } \\
\text{ end } \\
\text{ else vxvc_flag ← IsQNaN(src1) | IsQNaN(src2)} \\
\text{ if| CompareGTSP(src1,src2) | then do } \\
\text{ result{i:i+31} ← 0xFFFF_FFFF } \\
\text{ all_false ← 0b0 } \\
\text{ end } \\
\text{ else do } \\
\text{ result{i:i+31} ← 0x0000_0000 } \\
\text{ all_true ← 0b0 } \\
\text{ end } \\
\text{ if| vxsnan_flag then SetFX(VXSNAN) } \\
\text{ if| vxvc_flag then SetFX(VXVC) } \\
\text{ ex_flag ← ex_flag | (IVE & vxsnan_flag) } \\
\text{ ex_flag ← ex_flag | (IVE & vxvc_flag) } \\
\text{ end } \\
\text{ if| ex_flag = 0 ) then VSR[XT] ← result } \\
\text{ if| Rc=0| then do } \\
\text{ if| vex_flag then } \\
\text{ CR[6] ← all_true || 0b0 || all_false || 0b0 } \\
\text{ else } \\
\text{ CR[6] ← 0bUUUU } \\
\text{ end } \\
\text{ Let XT be the value } 32 \times TX + T. \\
\text{ Let XA be the value } 32 \times AX + A. \\
\text{ Let XB be the value } 32 \times BX + B. \\
\text{ For each vector element } i \text{ from } 0 \text{ to } 3, \text{ do the following. } \\
\text{ Let } src1 \text{ be the single-precision floating-point operand in word element } i \text{ of } VSR[XA]. \\
\text{ Let } src2 \text{ be the single-precision floating-point operand in word element } i \text{ of } VSR[XB]. \\
\text{ src1 is compared to src2. }
\end{array}
\]

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 of CR[6] is set to indicate all vector elements compared true.
- Bit 1 of CR[6] is set to 0.
- Bit 2 of CR[6] is set to indicate all vector elements compared false.
- Bit 3 of CR[6] 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[6] are undefined if \( Rc \) is equal to 1.

Special Registers Altered

<table>
<thead>
<tr>
<th>CR[6]</th>
<th>FX</th>
<th>VXSNAN</th>
<th>VXVC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

VSR Data Layout for xvcmpgtsp[]

\[
\begin{array}{cccccc}
\text{src1} & = & VSR[XA] \\
\text{src2} & = & VSR[XB] \\
\text{tgt} & = & VSR[XT] \\
\end{array}
\]

\[
\begin{array}{cccccc}
\text{SP} & \text{SP} & \text{SP} & \text{SP} \\
\text{SP} & \text{SP} & \text{SP} & \text{SP} \\
\text{MW} & \text{MW} & \text{MW} & \text{MW} \\
\end{array}
\]

670 Power ISA™ I
VSX Vector Copy Sign Double-Precision

**XX3-form**

\[
\begin{align*}
\text{xvcpsgndp} & \quad \text{XT,XA,XB} \\
\end{align*}
\]

**Table 100:**

<table>
<thead>
<tr>
<th></th>
<th>T</th>
<th>A</th>
<th>B</th>
<th>240</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
\text{XT} & \leftarrow \text{TX} || \text{T} \\
\text{XA} & \leftarrow \text{AX} || \text{A} \\
\text{XB} & \leftarrow \text{BX} || \text{B} \\
\end{align*}
\]

\[
\text{do} \quad \text{i} = 0 \text{ to } 127 \text{ by } 64 \\
\quad \text{VSR[XT]}^{[\text{i}:\text{i+63}]} \leftarrow \text{VSR[XA]}^{[\text{i}]} || \text{VSR[XB]}^{[\text{i+1}:\text{i+63}]} \\
\end{align*}
\]

Let XT be the value \(32 \times \text{TX} + \text{T}\).
Let XA be the value \(32 \times \text{AX} + \text{A}\).
Let XB be the value \(32 \times \text{BX} + \text{B}\).

For each vector element i from 0 to 1, do the following.
The contents of bit 0 of doubleword element i of VSR[XA] are concatenated with the contents of bits 1:63 of doubleword element i of VSR[XB] and placed into doubleword element i of VSR[XT].

**Special Registers Altered**

None

**Extended Mnemonic Equivalent To**

\[
\begin{align*}
\text{XVMOVD} & \quad \text{XT, XB} \\
\text{XVCPSGNDP} & \quad \text{XT, XB, XB} \\
\end{align*}
\]

VSR Data Layout for xvcpsgndp

src1 = VSR[XA]

<table>
<thead>
<tr>
<th></th>
<th>DP</th>
<th>DP</th>
</tr>
</thead>
</table>

src2 = VSR[XB]

<table>
<thead>
<tr>
<th></th>
<th>DP</th>
<th>DP</th>
</tr>
</thead>
</table>

tgt = VSR[XT]

<table>
<thead>
<tr>
<th></th>
<th>64</th>
<th>127</th>
</tr>
</thead>
</table>

VSX Vector Copy Sign Single-Precision

**XX3-form**

\[
\begin{align*}
\text{xvcpsgnsp} & \quad \text{XT,XA,XB} \\
\end{align*}
\]

**Table 101:**

<table>
<thead>
<tr>
<th></th>
<th>T</th>
<th>A</th>
<th>B</th>
<th>208</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
\text{XT} & \leftarrow \text{TX} || \text{T} \\
\text{XA} & \leftarrow \text{AX} || \text{A} \\
\text{XB} & \leftarrow \text{BX} || \text{B} \\
\end{align*}
\]

\[
\text{do} \quad \text{i} = 0 \text{ to } 127 \text{ by } 32 \\
\quad \text{VSR[XT]}^{[\text{i}:\text{i+31}]} \leftarrow \text{VSR[XA]}^{[\text{i}]} || \text{VSR[XB]}^{[\text{i+1}:\text{i+31}]} \\
\end{align*}
\]

Let XT be the value \(32 \times \text{TX} + \text{T}\).
Let XA be the value \(32 \times \text{AX} + \text{A}\).
Let XB be the value \(32 \times \text{BX} + \text{B}\).

For each vector element i from 0 to 3, do the following.
The contents of bit 0 of word element i of VSR[XA] are concatenated with the contents of bits 1:31 of word element i of VSR[XB] and placed into word element i of VSR[XT].

**Special Registers Altered**

None

**Extended Mnemonic Equivalent To**

\[
\begin{align*}
\text{XVMOvsp} & \quad \text{XT, XB} \\
\text{XVCPSGNSP} & \quad \text{XT, XB, XB} \\
\end{align*}
\]

VSR Data Layout for xvcpsgnsp

src1 = VSR[XA]

<table>
<thead>
<tr>
<th></th>
<th>SP</th>
<th>SP</th>
<th>SP</th>
<th>SP</th>
</tr>
</thead>
</table>

src2 = VSR[XB]

<table>
<thead>
<tr>
<th></th>
<th>SP</th>
<th>SP</th>
<th>SP</th>
<th>SP</th>
</tr>
</thead>
</table>

tgt = VSR[XT]

<table>
<thead>
<tr>
<th></th>
<th>32</th>
<th>64</th>
<th>96</th>
<th>127</th>
</tr>
</thead>
</table>

Chapter 7. Vector-Scalar Floating-Point Operations 671
VSX Vector Convert with round
Double-Precision to Single-Precision format
XX2-form

xvcvdpsp XT,XB

<table>
<thead>
<tr>
<th>60</th>
<th>393</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>B</td>
</tr>
</tbody>
</table>

Let XT be the value $32 \times TX + T$.
Let XB be the value $32 \times BX + B$.

For each vector element $i$ 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 of doubleword element $i$ of VSR[XT] in single-precision format.

The contents of bits 32:63 of doubleword element $i$ of VSR[XT] are undefined.

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

VSR Data Layout for xvcvdpsp

src = VSR[XB]

<table>
<thead>
<tr>
<th></th>
<th>DP</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

tgt = VSR[XT]

<table>
<thead>
<tr>
<th></th>
<th>SP</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>undefined</td>
<td></td>
</tr>
</tbody>
</table>

0 64 96 127
VSX Vector Convert with round to zero
Double-Precision to Signed Doubleword format XX2-form

\[
\begin{array}{ccccccc}
\text{xvcvdpsxds} & \text{XT, XB} & \text{0} & \text{60} & \text{T} & \text{11} & \text{B}
\end{array}
\]

\[
\begin{array}{ccccccc}
\text{VSR Data Layout for xvcvdpsxds} & \text{src = VSR[XB]} & \text{DP} & \text{DP} & \text{tgt = VSR[XT]} & \text{SD} & \text{SD}
\end{array}
\]

Programming Note

\text{xvcvdpsxds} rounds using 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.

Let \( XT \) be the value \( 32 \times TX + T \).
Let \( XB \) be the value \( 32 \times BX + B \).

For each vector element \( i \) from 0 to 1, do the following.

Let \( src \) be the double-precision floating-point operand in doubleword element \( i \) 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 \( 2^{63} \), 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 102.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].
### Table 102. Actions for xvcvdpsxds

<table>
<thead>
<tr>
<th>src</th>
<th>Nmin-1</th>
<th>Nmin-1 &lt; src &lt; Nmin</th>
<th>Nmin = src</th>
<th>Nmin &lt; src &lt; Nmax</th>
<th>Nmax = src</th>
<th>Nmax &lt; src &lt; Nmax+1</th>
<th>src is a QNaN</th>
<th>src is a SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>0</td>
<td>yes</td>
<td>T[min], fx(VXCVI)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>1</td>
<td>yes</td>
<td>fx(VXCVI), error()</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
<td>no</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>0</td>
<td>yes</td>
<td>T[max], fx(VXCVI)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>1</td>
<td>yes</td>
<td>fx(VXCVI), error()</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
<td>no</td>
<td>T[max]</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>0</td>
<td>yes</td>
<td>T[max], fx(VXCVI)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>1</td>
<td>yes</td>
<td>fx(VXCVI), error()</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
<td>no</td>
<td>T[max]</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>0</td>
<td>yes</td>
<td>T[min], fx(VXCVI)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>1</td>
<td>yes</td>
<td>fx(VXCVI), error()</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
<td>no</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>0</td>
<td>yes</td>
<td>T[min], fx(VXCVI)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>1</td>
<td>yes</td>
<td>fx(VXCVI), error()</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
<td>no</td>
<td>T[min]</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

**Explanation:**

- **fx(x)**: FX is set to 1 if x=0. x is set to 1.
- **error()**: The system error handler is invoked for the trap-enabled exception if the FE0 and FE1 bits in the Machine State Register are set to any mode other than the ignore-exception mode. Update of VSR[XT] is suppressed.
- **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 i of VSR[XB] (where i ∈ {0, 1}).
- **T(x)**: The signed integer doubleword value x is placed in doubleword element i of VSR[XT] (where i ∈ {0, 1}).
VSX Vector Convert with round to zero
Double-Precision to Signed Word format
XX2-form

```
xvcvdpsxs  XT,XB

<table>
<thead>
<tr>
<th>60</th>
<th>6</th>
<th>T</th>
<th>//</th>
<th>B</th>
<th>216</th>
<th>BITX</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>T</td>
<td>//</td>
<td>B</td>
<td>216</td>
<td>BITX</td>
</tr>
</tbody>
</table>
```

Let $XT$ be the value $32 \times TX + T$.
Let $XB$ be the value $32 \times BX + B$.

For each vector element $i$ from 0 to 1, do the following.
Let $src$ be the double-precision floating-point operand in doubleword 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 Toward Zero.

If the rounded value is greater than $2^{31} \cdot 1$, the result is $0x7FFFFFFF$ 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 bits 0:31 of doubleword element $i$ of $VSR[XT]$.

The contents of bits 32:63 of doubleword element 1 of $VSR[XT]$ are undefined.

See Table 103.
Inexact? (RoundToDPintegerTrunc(src))

Returned Results and Status Setting

<table>
<thead>
<tr>
<th>src</th>
<th>Nmin-1</th>
<th>Nmin-1 &lt; src &lt; Nmin</th>
<th>src = Nmin</th>
<th>Nmin &lt; src &lt; Nmax</th>
<th>src = Nmax</th>
<th>Nmax &lt; src &lt; Nmax+1</th>
<th>src is a QNaN</th>
<th>src is a SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<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>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>yes</td>
<td>Nmin</td>
<td>-</td>
<td>no</td>
<td>T(ConvertDPtoSW(RoundToDPintegerTrunc(src)))</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>yes</td>
<td>Nmin</td>
<td>-</td>
<td>no</td>
<td>T(ConvertDPtoSW(RoundToDPintegerTrunc(src)))</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>no</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>T(max)</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>no</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>T(max)</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>no</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>yes</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>no</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>yes</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>T(min)</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>yes</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>T(min)</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>yes</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Explanation:

- fx(x): FX is set to 1 if x=0. x is set to 1.
- error(): The system error handler is invoked for the trap-enabled exception if the FE0 and FE1 bits in the Machine State Register are set to any mode other than the ignore-exception mode.
- Update of VSR[XT] is suppressed.
- 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 double-precision floating-point value in doubleword element i of VSR[XB] (where i \(\in \{0,1\}\)).
- T(x): The signed integer word value x is placed in word element i of VSR[XT] (where i \(\in \{0,2\}\)).

Table 103. Actions for xvcvdpsxws
VSX Vector Convert with round to zero
Double-Precision to Unsigned Doubleword format XX2-form

\[ \text{xcvdpuxds} \quad \text{XT, XB} \]


<table>
<thead>
<tr>
<th>60</th>
<th>56</th>
<th>52</th>
<th>48</th>
<th>44</th>
<th>36</th>
<th>32</th>
<th>28</th>
<th>24</th>
<th>20</th>
<th>16</th>
<th>12</th>
<th>8</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>//</td>
<td>B</td>
<td>//</td>
<td>456</td>
<td>0</td>
<td>0</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>30</td>
<td>31</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

\[
\text{XT} \leftarrow \text{TX} || \text{T} \\
\text{XB} \leftarrow \text{BX} || \text{B} \\
\text{ex\_flag} \leftarrow 0000_0000_0000_0000
\]

do i = 0 to 127 by 64
    reset_xflags();
    result\{i:i+63\} \leftarrow \text{ConvertDPtoUD(VSR[XB]{i:i+63})}
    if(xvsnan\_flag) then SetFX(VXSNAN)
    if(vxcvi\_flag) then SetFX(VXCVI)
    if(xx\_flag) then SetFX(XX)
    ex\_flag \leftarrow \text{ex\_flag | (VE & xvsnan\_flag)}
    ex\_flag \leftarrow \text{ex\_flag | (VE & vxcvi\_flag)}
    ex\_flag \leftarrow \text{ex\_flag | (XE & xx\_flag)}
end

if( ex\_flag = 0 ) then VSR[XT] \leftarrow \text{result}

Let XT be the value \(32 \times \text{TX} + \text{T}\).

Let XB be the value \(32 \times \text{BX} + \text{B}\).

For each vector element \(i\) from 0 to 1, do the following.

Let src be the double-precision floating-point operand in doubleword element \(i\) 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 \(0xFFF_FFFF_FFFF_FFFF\) 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 104.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].
### Returned Results and Status Setting

<table>
<thead>
<tr>
<th>src</th>
<th>NE</th>
<th>FC</th>
<th>Result</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>fx(VXCVI), error()</td>
</tr>
<tr>
<td>Nmin-1 &lt; src &lt; Nmin</td>
<td>-</td>
<td>0</td>
<td>T(Nmin), fx(XX)</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>yes</td>
<td>fx(XX), 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>0</td>
<td>T(ConvertDPtoUD(RoundToDPintegerTrunc(src)))</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>yes</td>
<td>fx(XX), error()</td>
</tr>
<tr>
<td>src = Nmax</td>
<td>-</td>
<td>-</td>
<td>no</td>
</tr>
<tr>
<td>Nmax &lt; src &lt; Nmax+1</td>
<td>-</td>
<td>0</td>
<td>T(Nmax), fx(XX)</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>yes</td>
<td>T(Nmax), fx(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>fx(VXCVI), error()</td>
</tr>
<tr>
<td>src is a QNaN</td>
<td>0</td>
<td>-</td>
<td>T(Nmin), fx(VXCVI), fx(VXSNAN)</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>-</td>
<td>fx(VXCVI), fx(VXSNAN), 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>fx(VXCVI), fx(VXSNAN), error()</td>
</tr>
</tbody>
</table>

**Explanation:**

- **fx(x)**: FX is set to 1 if x=0, x is set to 1.
- **error()**: The system error handler is invoked for the trap-enabled exception if the FE0 and FE1 bits in the Machine State Register are set to any mode other than the ignore-exception mode. Update of VSR[XT] is suppressed.
- **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 \(i\) VSR[XB] (where \(i \in \{0, 1\}\)).
- **T(x)**: The unsigned integer doubleword value \(x\) is placed in doubleword element \(i\) of VSR[XT] (where \(i \in \{0, 1\}\)).

**Table 104. Actions for xvcvdpuxds**
VSX Vector Convert with round to zero
Double-Precision to Unsigned Word format
XX2-form

```
xvcvdpuxws  XT,XB

<table>
<thead>
<tr>
<th>60</th>
<th>6</th>
<th>T</th>
<th>11</th>
<th>B</th>
<th>200</th>
<th>DTX</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

XT ← TX || T
XB ← BX || B
ex_flag ← 000

do i=0 to 127 by 64
  reset_xflags()
  result{i:i+31} ← ConvertDPtoUW(VSR[XB]{i:i+63})
  result{i+32:i+63} ← 0xUUUU_UUUU
  if(vxsnan_flag) then SetFX(VXSNAN)
  if(vxcvi_flag)  then SetFX(VXCVI)
  if(xx_flag)     then SetFX(XX)
  ex_flag ← ex_flag | (VE & vxsnan_flag)
  ex_flag ← ex_flag | (VE & vxcvi_flag)
  ex_flag ← ex_flag | (XE & xx_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

For each vector element i from 0 to 1, do the following.
  Let src be the double-precision floating-point operand in doubleword 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 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.

  The result is placed into bits 0:31 of doubleword element i of VSR[XT].

  The contents of bits 32:63 of doubleword element i of VSR[XT] are undefined.

See Table 105.

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

VSR Data Layout for xvcvdpuxws
src = VSR[XB]
tgt = VSR[XT]

<table>
<thead>
<tr>
<th>UW</th>
<th>undefined</th>
<th>UW</th>
<th>undefined</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>32</td>
<td>64</td>
<td>96</td>
</tr>
</tbody>
</table>

Programming Note

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 xvrdpic which uses the rounding mode specified by RN.
### Returned Results and Status Setting

<table>
<thead>
<tr>
<th>src</th>
<th>NE</th>
<th>Impact? (RoundToDPIntegerTrunc(src) &amp; src)</th>
<th>Returned Results and Status Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nmin-1</td>
<td>0</td>
<td>-</td>
<td>T[min], fX(VXCVI)</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>-</td>
<td>fX(VXCVI), error()</td>
</tr>
<tr>
<td>Nmin-1 &lt; src &lt; Nmin</td>
<td>-</td>
<td>0</td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>yes</td>
<td>fX(XX), 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>-</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>yes</td>
<td>T(ConvertDPtoUW(RoundToDPIntegerTrunc(src))), fX(XX)</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>yes</td>
<td>fX(XX), error()</td>
</tr>
<tr>
<td>src = Nmax</td>
<td>-</td>
<td>-</td>
<td>no</td>
</tr>
<tr>
<td>Nmax &lt; src &lt; Nmax+1</td>
<td>-</td>
<td>0</td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>yes</td>
<td>fX(XX), error()</td>
</tr>
<tr>
<td>src = Nmax+1</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>src is a QNaN</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>src is a SNaN</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

**Explanation:**

- **fx(x)**  
  FX is set to 1 if x=0. x is set to 1.
- **error()**  
  The system error handler is invoked for the trap-enabled exception if the FE0 and FE1 bits in the Machine State Register are set to any mode other than the ignore-exception mode.
- **Update of VSR[XT]**  
  Update of VSR[XT] is suppressed.
- **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 in doubleword element i of VSR[XB] (where i ∈ {0,1}).
- **T(x)**  
  The unsigned integer word value x is placed in word element i of VSR[XT] (where i ∈ {0,2}).

---

**Table 105. Actions for xvcvdpuxws**
VSX Vector Convert Half-Precision to Single-Precision format XX2-form

\[
\text{VSVX Vector Convert Half-Precision to Single-Precision format XX2-form}
\]

\[
\text{xvcvhpsp XT,XB}
\]

\[
\begin{array}{cccc}
60 & 24 & B & 475 \\
\end{array}
\]

\[
\begin{array}{cccc}
T & 11 & 16 & p1 \\
\end{array}
\]

\[
\begin{array}{cccc}
B[X] & 60 & T & 11 \\
\end{array}
\]

\[
\begin{array}{cccc}
B & 475 & B[X] & 60 \\
\end{array}
\]

\[
\begin{array}{cccc}
0 & 6 & 11 & 16 \\
21 & 30 & 31 \\
\end{array}
\]

\[
\begin{array}{cccc}
if \text{MSR.VSX}=0 \text{ then VSX.Unavailable}() \\
\text{reset_flags()} \\
do i = 0 \text{ to } 3 \\
\text{src} \leftarrow \text{bfp_CONVERT_FROM_BFP16(VSR}[BX]\text{word[i].hword[1]])} \\
\text{if src.class.SNaN=1 then} \\
\text{result.word[i] \leftarrow bfp_CONVERT_TO_BFP32(bfp_QUIET(src))} \\
\text{else} \\
\text{result.word[i] \leftarrow bfp_CONVERT_TO_BFP32(src)} \\
\text{vxsnan_flag} \leftarrow \text{src.class.SNaN} \\
\text{if(vxsnan_flag) then SetFX(FPSCR.VXSNAN)} \\
\text{ex_flag} \leftarrow \text{ex_flag | (FPSCR.VE & vxsnan_flag)} \\
\text{end} \\
\text{if ex_flag=0 then VSR}[XT] \leftarrow \text{result}
\]

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 half-precision floating-point value in the rightmost halfword of word element i of VSR[XB].

If src is an SNaN, the result is the single-precision representation of that SNaN converted to a QNaN.

Otherwise, if src is a QNaN, the result is the single-precision representation of that QNaN.

Otherwise, if src is an Infinity, the result is the single-precision representation of Infinity with the same sign as src.

Otherwise, if src is a Zero, the result is the single-precision representation of Zero with the same sign as src.

Otherwise, if src is a denormal value, the result is the normalized single-precision representation of src.

Otherwise, the result is the single-precision representation of src.

The result is placed into word element i of VSR[XT].

If a trap-enabled exception occurs, VSR[XT] is not modified.

Special Registers Altered:

FX VXSNAN

VSR Data Layout for xvcvhpsp

\[
\begin{array}{cccc}
s \text{src} & \text{unused} & \text{VSR}[X].\text{word}[0] & \text{unused} \\
\text{tgt} & \text{VSR}[X].\text{word}[1] & \text{VSR}[X].\text{word}[2] & \text{VSR}[X].\text{word}[3] \\
0 & 16 & 32 & 48 \\
64 & 80 & 96 & 112 \\
127 \\
\end{array}
\]
### VSX Vector Convert Single-Precision to Double-Precision format XX2-form

**xvcvspdp** XT,XB

<table>
<thead>
<tr>
<th>0</th>
<th>60</th>
<th>T</th>
<th>(///)</th>
<th>B</th>
<th>457</th>
<th>XTX</th>
</tr>
</thead>
</table>

\[
\begin{align*}
XT & \leftarrow TX || T \\
XB & \leftarrow BX || B \\
ex\_flag & \leftarrow 0b0 \\
do & \text{ i=0 to } 127 \text{ by } 64 \\
\quad & \text{ reset\_flags() } \\
\quad & \text{ result}[(i:i+63) \leftarrow \text{ ConvertSPtoDP(VSR[XB][i:i+31])} \\
\quad & \text{ if(vxsnan\_flag) then SetFX(VXSNAN)} \\
\quad & \text{ ex\_flag \leftarrow ex\_flag | (VE \& vxsnan\_flag)} \\
\end{align*}
\]

\[
\begin{align*}
\text{end} \\
\text{ if( ex\_flag = 0 ) then VSR[XT] \leftarrow \text{ result} }
\end{align*}
\]

Let XT be the value \(32 \times TX + T\).
Let XB be the value \(32 \times BX + B\).

For each vector element \(i\) 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].

\[
\text{src is placed into doubleword element }i\text{ of VSR[XT] in double-precision format.}
\]

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

**src = VSR[XB]**

<table>
<thead>
<tr>
<th>SP</th>
<th>unused</th>
<th>SP</th>
<th>unused</th>
</tr>
</thead>
</table>

**tgt = VSR[XT]**

<table>
<thead>
<tr>
<th>DP</th>
<th></th>
<th>64</th>
<th>96</th>
<th>127</th>
</tr>
</thead>
</table>
Section 7. Vector-Scalar Floating-Point Operations

### VSX Vector Convert with round

**Single-Precision to Half-Precision 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 value in word element \( i \) 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 word element \( i \) of \( VSR[XT] \).

If a trap-enabled exception occurs, \( VSR[XT] \) is not modified.

**Special Registers Altered:**

- FX VXSNAN OX UX XX

---

### VSR Data Layout for xvcvsphp

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>tgt</td>
<td>0x0000</td>
<td>VSR[XT].hword[0]</td>
<td>0x0000</td>
<td>VSR[XT].hword[1]</td>
</tr>
</tbody>
</table>

---
VSX Vector Convert with round to zero
Single-Precision to Signed Doubleword format XX2-form

\[
xvcvpsxds \quad XT, XB
\]

<table>
<thead>
<tr>
<th>( s )</th>
<th>( T )</th>
<th>( /// )</th>
<th>( B )</th>
<th>( 408 )</th>
<th>( \text{X} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>60</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>0</td>
</tr>
</tbody>
</table>

explain the code:

Let \( XT \) be the value \( 32xTX + T \).
Let \( XB \) be the value \( 32xBX + B \).

For each vector element \( i \) from 0 to 1, do the following.

Let \( src \) be the single-precision floating-point operand in word element \( i x2 \) 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 \( 2^{63} \), 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 105.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[\( XT \)].
### Returned Results and Status Setting

<table>
<thead>
<tr>
<th>src</th>
<th>Nmin-1</th>
<th>Nmin-1 &lt; src &lt; Nmin</th>
<th>src = Nmin</th>
<th>Nmin &lt; src &lt; Nmax</th>
<th>src = Nmax</th>
<th>Nmax &lt; src &lt; Nmax+1</th>
<th>src is a QNaN</th>
<th>src is a SNaN</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>
<tr>
<td>1</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>0</td>
<td>yes</td>
<td>T(Nmin) fx(XX)</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>1</td>
<td>yes</td>
<td>T(Nmin) fx(XX)</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>–</td>
<td>–</td>
<td>no</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>–</td>
<td>–</td>
<td>no</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>

Note: This case cannot occur as Nmax is not representable in SP format but is included here for completeness.

### Explanation:

- $f(x)$: $F_X$ is set to 1 if $x=0$. $x$ is set to 1.
- $error()$: The system error handler is invoked for the trap-enabled exception if the $FE0$ and $FE1$ bits in the Machine State Register are set to any mode other than the ignore-exception mode. Update of VSR[XT] is suppressed.
- $N_{min}$: The smallest signed integer doubleword value, $-2^{63}$ (0x8000_0000_0000_0000).
- $N_{max}$: The largest signed integer doubleword value, $2^{63} - 1$ (0x7FFF_FFFF_FFFF_FFFF).
- src: The single-precision floating-point value in word element $i$ of VSR[XB] (where $i \in \{0, 2\}$).
- $T(x)$: The signed integer doubleword value $x$ is placed in doubleword element $i$ of VSR[XT] (where $i \in \{0, 1\}$).

### Table 106. Actions for xvcvpsxds
VSX Vector Convert with round to zero
Single-Precision to Signed Word format
XX2-form

xcvpsxws XT, XB

| src = VSR[XB] |
|---|---|---|---|---|
| SP | SP | SP | SP |

tgt = VSR[XT]

| tgt = VSR[XT] |
|---|---|---|---|---|
| SW | SW | SW | SW |

Programming Note

xcvpsxws 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.

Special Registers Altered

FX XX VXSNAN VXCVI

Let XT be the value 32xTX + T.
Let XB be the value 32xBX + B.

For each vector element 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 Toward 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 105.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].
## Table 107. Actions for xvcvspbxws

<table>
<thead>
<tr>
<th>src [ Nmin-1</th>
<th></th>
<th></th>
<th>Returned Results and Status Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nmin-1 &lt; src &lt; Nmin</td>
<td>-</td>
<td>0</td>
<td>T(Nmin), fx(VXCVI)</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>1</td>
<td>fx(XX), error()</td>
</tr>
<tr>
<td>src = Nmin</td>
<td>-</td>
<td>-</td>
<td>no T(Nmin)</td>
</tr>
<tr>
<td>Nmin &lt; src &lt; Nmax</td>
<td>-</td>
<td>0</td>
<td>T(ConvertSPtoSW(RoundToSPIntegerTrunc(src))), fx(XX)</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>1</td>
<td>fx(XX), error()</td>
</tr>
<tr>
<td>src = Nmax</td>
<td>-</td>
<td>-</td>
<td>no T(Nmax)</td>
</tr>
<tr>
<td></td>
<td>Note: This case cannot occur as Nmax is not representable in SP format but is included here for completeness.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nmax &lt; src &lt; Nmax+1</td>
<td>-</td>
<td>0</td>
<td>T(Nmax), fx(XX)</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>1</td>
<td>fx(XX), error()</td>
</tr>
<tr>
<td>src in Nmax+1</td>
<td>0</td>
<td>-</td>
<td>T(Nmax), fx(VXCVI)</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>-</td>
<td>fx(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>fx(VXCVI), fx(VXSNAN), 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>fx(VXCVI), fx(VXSNAN), error()</td>
</tr>
</tbody>
</table>

### Explanation:

- **fx(x)**: FX is set to 1 if x=0. x is set to 1.
- **error()**: The system error handler is invoked for the trap-enabled exception if the FE0 and FE1 bits in the Machine State Register are set to any mode other than the ignore-exception mode. Update of VSR[XT] is suppressed.
- **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 x is placed in word element i of VSR[XT] (where i ∈ {0,1,2,3}).

---

Table 107. Actions for xvcvspbxws
VSX Vector Convert with round to zero
Single-Precision to Unsigned Doubleword
format XX2-form

xvcvspuxds XT,XB

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

For each vector element i from 0 to 1, do the following.
Let src be the single-precision floating-point operand in word element i×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 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.

The result is placed into doubleword element i of VSR[XT].

See Table 105.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].
### Returned Results and Status Setting

<table>
<thead>
<tr>
<th>src</th>
<th>VE</th>
<th>XE</th>
<th>T(x)</th>
<th>fx(x)</th>
<th>error()</th>
</tr>
</thead>
<tbody>
<tr>
<td>src ≤ Nmin</td>
<td>0</td>
<td>–</td>
<td>–</td>
<td>T(Nmin)</td>
<td>fx(VXCVI)</td>
</tr>
<tr>
<td>1</td>
<td>–</td>
<td>–</td>
<td>fx(VXCVI), error()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nmin+1 &lt; src &lt; Nmin</td>
<td>–</td>
<td>0</td>
<td>yes</td>
<td>T(Nmin)</td>
<td>fx(XX)</td>
</tr>
<tr>
<td>1</td>
<td>yes</td>
<td>fx(XX), error()</td>
<td></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>–</td>
<td>no</td>
<td>T(ConvertSPtoUD(RoundToSPIntegerTrunc(src)))</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>yes</td>
<td>T(ConvertSPtoUD(RoundToSPIntegerTrunc(src))), fx(XX)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>yes</td>
<td>fx(XX), error()</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>src = Nmax</td>
<td>–</td>
<td>–</td>
<td>no</td>
<td>T(Nmax)</td>
<td></td>
</tr>
<tr>
<td>Note: This case cannot occur as Nmax is not representable in SP format but is included here for completeness.</td>
<td></td>
<td></td>
<td></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)</td>
<td>fx(XX)</td>
</tr>
<tr>
<td>1</td>
<td>yes</td>
<td>fx(XX), error()</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>src in Nmax+1</td>
<td>0</td>
<td>–</td>
<td>–</td>
<td>T(Nmax)</td>
<td>fx(VXCVI)</td>
</tr>
<tr>
<td>1</td>
<td>–</td>
<td>–</td>
<td>fx(VXCVI), error()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>src is a QNaN</td>
<td>0</td>
<td>–</td>
<td>–</td>
<td>T(Nmin)</td>
<td>fx(VXCVI), fx(VXSNAN)</td>
</tr>
<tr>
<td>1</td>
<td>–</td>
<td>–</td>
<td>fx(VXCVI), fx(VXSNAN), error()</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Explanation:**
- **fx(x)**: FX is set to 1 if x=0. x is set to 1.
- **error()**: The system error handler is invoked for the trap-enabled exception if the FE0 and FE1 bits in the Machine State Register are set to any mode other than the ignore-exception mode.
- **Update of VSR[XT]**: is suppressed.

- **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 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}).

---

**Table 108. Actions for xvcvspuxds**
### VSX Vector Convert with round to zero
### Single-Precision to Unsigned Word format
### XX2-form

#### xvcvspuxws

<table>
<thead>
<tr>
<th>0</th>
<th>60</th>
<th>T</th>
<th>11</th>
<th>B</th>
<th>136</th>
<th>15</th>
<th>21</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**VT** ← TX || T
**VB** ← BX || B
**ex_flag** ← 0b0

do i=0 to 127 by 32
  reset_exflags()
  result{i:i+31} ← ConvertSPtoUW(VSR[XB][i:i+31])
  if(vxsnan_flag) then SetFX(VXSNAN)
  if(vxcvi_flag)  then SetFX(VXCVI)
  if(xx_flag)     then SetFX(XX)
  ex_flag ← ex_flag | (VE & vxsnan_flag)
  ex_flag ← ex_flag | (VE & vxcvi_flag)
  ex_flag ← ex_flag | (XE & xx_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result

Let XT be the value 32xTX + T.
Let XB be the value 32xBX + B.

For each vector element 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-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 105.

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.

---

#### VSR Data Layout for xvcvspuxws

| src = VSR[XB] |
|---|---|---|---|
| SP | SP | SP | SP |

tgt = VSR[XT]

| tgt = VSR[XT] |
|---|---|---|---|
| UW | UW | UW | UW |

---

#### Table 105

| VSR Data Layout for xvcvspuxws |
|---|---|---|---|
| src = VSR[XB] |
|---|---|---|---|
| SP | SP | SP | SP |

tgt = VSR[XT]

| tgt = VSR[XT] |
|---|---|---|---|
| UW | UW | UW | UW |
### Returned Results and Status Setting

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Nmin-1</td>
<td>0 -</td>
<td>T(Nmin), fx(VXCVI)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nmin-1</td>
<td>1</td>
<td>-</td>
<td>fx(VXCVI), error()</td>
<td></td>
</tr>
<tr>
<td>Min-1 &lt; src &lt; Nmin</td>
<td>- 0</td>
<td>yes</td>
<td>T(Nmin), fx(XX)</td>
<td></td>
</tr>
<tr>
<td>Min-1 &lt; src &lt; Nmin</td>
<td>1</td>
<td>yes</td>
<td>fx(XX), error()</td>
<td></td>
</tr>
<tr>
<td>src = Nmin</td>
<td>- -</td>
<td>no</td>
<td>T(Nmin)</td>
<td></td>
</tr>
<tr>
<td>Min &lt; src &lt; Nmax</td>
<td>- - no</td>
<td>T(ConvertSPtoUW(RoundToSPintegerTrunc(src)))</td>
<td></td>
<td></td>
</tr>
<tr>
<td>src = Nmax</td>
<td>- - no</td>
<td>T(Nmax)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Note: This case cannot occur as Nmax is not representable in SP format but is included here for completeness.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nmax &lt; src &lt; Nmax+1</td>
<td>- 0</td>
<td>yes</td>
<td>T(Nmax), fx(XX)</td>
<td></td>
</tr>
<tr>
<td>Nmax &lt; src &lt; Nmax+1</td>
<td>1</td>
<td>yes</td>
<td>fx(XX), error()</td>
<td></td>
</tr>
<tr>
<td>src = Nmax+1</td>
<td>0 -</td>
<td>T(Nmax), fx(VXCVI)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>src = Nmax+1</td>
<td>1</td>
<td>-</td>
<td>fx(VXCVI), error()</td>
<td></td>
</tr>
<tr>
<td>src is a QNaN</td>
<td>0 -</td>
<td>T(Nmin), fx(VXCVI), fx(VXSNAN)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>src is a SNaN</td>
<td>1 -</td>
<td>-</td>
<td>fx(VXCVI), fx(VXSNAN), error()</td>
<td></td>
</tr>
</tbody>
</table>

### Explanation:
- **fx(x)**: FX is set to 1 if x=0. x is set to 1.
- **error()**: The system error handler is invoked for the trap-enabled exception if the FE0 and FE1 bits in the Machine State Register are set to any mode other than the ignore-exception mode.
- **Update of VSR[XT]** is suppressed.
- **Nmin**: The smallest unsigned integer word value, 0x0000_0000.
- **Nmax**: The largest unsigned integer word value, 2^{32}-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}).

### Table 109. Actions for xvcvspuxws
**VSX Vector Convert with round Signed Doubleword to Double-Precision format**

### xx2-form

**xvcvsxddp**  
**XT, XB**

<p>| | | | | | | | | | | | |</p>
<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>
</tr>
</thead>
<tbody>
<tr>
<td>60</td>
<td>T</td>
<td></td>
<td></td>
<td>B</td>
<td>21</td>
<td>504</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
\begin{align*}
\text{XT} & \leftarrow TX || T \\
\text{XB} & \leftarrow BX || B \\
\text{ex_flag} & \leftarrow 0b0 \\
\end{align*}
\]

\[
\begin{align*}
\text{do } & \text{i=0 to 127 by 64} \\
\text{reset_xflags()} \\
\text{v[0:inf]} & \leftarrow \text{ConvertSDtoFP(VSR[XB]{i:i+63})} \\
\text{result[i:i+63]} & \leftarrow \text{RoundToDP(RN,v)} \\
\text{if(ex_flag)} & \text{then SetFX(XX)} \\
\text{ex_flag} & \leftarrow \text{ex_flag} | (XE \& xx_flag) \\
\text{end} \\
\text{if( ex_flag = 0 ) then VSR[XT] } & \leftarrow \text{result} \\
\end{align*}
\]

Let \( XT \) be the value \( 32 \times TX + T \).

Let \( XB \) be the value \( 32 \times BX + B \).

For each vector element \( i \) 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 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**

\[
\begin{align*}
\text{src} & = \text{VSR[XB]} \\
\text{tgt} & = \text{VSR[XT]} \\
\end{align*}
\]

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>SD</td>
<td>SD</td>
<td>DP</td>
<td>DP</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
\begin{align*}
\text{0} & \quad 64 & \quad 127 \\
\end{align*}
\]

---

**VSX Vector Convert with round Signed Doubleword to Single-Precision format**

### xx2-form

**xvcvsxdsp**  
**XT, XB**

<p>| | | | | | | | | | | | |</p>
<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>
</tr>
</thead>
<tbody>
<tr>
<td>60</td>
<td>T</td>
<td></td>
<td></td>
<td>B</td>
<td>21</td>
<td>440</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
\begin{align*}
\text{XT} & \leftarrow TX || T \\
\text{XB} & \leftarrow BX || B \\
\text{ex_flag} & \leftarrow 0b0 \\
\end{align*}
\]

\[
\begin{align*}
\text{do } & \text{i=0 to 127 by 64} \\
\text{reset_xflags()} \\
\text{v[0:inf]} & \leftarrow \text{ConvertSDtoFP(VSR[XB]{i:i+63})} \\
\text{result[i:i+31]} & \leftarrow \text{RoundToSP(RN,v)} \\
\text{result[i+32:i+63]} & \leftarrow 0xUUUU_UUUU \\
\text{if(ex_flag)} & \text{then SetFX(XX)} \\
\text{ex_flag} & \leftarrow \text{ex_flag} | (XE \& xx_flag) \\
\text{end} \\
\text{if( ex_flag = 0 ) then VSR[XT] } & \leftarrow \text{result} \\
\end{align*}
\]

Let \( XT \) be the value \( 32 \times TX + T \).

Let \( XB \) be the value \( 32 \times BX + B \).

For each vector element \( i \) 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 contents of bits 32:63 of doubleword element \( i \) of VSR[XT] are undefined.

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 xvcvsxdsp**

\[
\begin{align*}
\text{src} & = \text{VSR[XB]} \\
\text{tgt} & = \text{VSR[XT]} \\
\end{align*}
\]

<p>| | | | | | | | | | | | |</p>
<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>
</tr>
</thead>
<tbody>
<tr>
<td>SD</td>
<td>SD</td>
<td>SP</td>
<td>undefined</td>
<td>SP</td>
<td>undefined</td>
<td>SP</td>
<td>undefined</td>
<td>SP</td>
<td>undefined</td>
<td>SP</td>
<td>undefined</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
\text{0} & \quad 32 & \quad 64 & \quad 96 & \quad 127 \\
\end{align*}
\]
### VSX Vector Convert Signed Word to Double-Precision format XX2-form

**VCVXSWDP** XT,XB

```plaintext
<table>
<thead>
<tr>
<th>60</th>
<th>T</th>
<th>///</th>
<th>B</th>
<th>248</th>
<th>bit</th>
<th>60</th>
</tr>
</thead>
</table>
```

```plaintext
do i = 0 to 1
  src ← bfp_CONVERT_FROM_SI32(VSR[32×BX+B].dword[i].word[0])
  VSR[32×TX+T].dword[i] ← bfp64_CONVERT_FROM_BFP(src)
end
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

For each vector element 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 VCVXSWDP**

<table>
<thead>
<tr>
<th>src = VSR[XB]</th>
</tr>
</thead>
<tbody>
<tr>
<td>SW</td>
</tr>
<tr>
<td>tgt = VSR[XT]</td>
</tr>
<tr>
<td>DP</td>
</tr>
</tbody>
</table>

### VSX Vector Convert with round Signed Word to Single-Precision format XX2-form

**VCVXWSP** XT,XB

```plaintext
<table>
<thead>
<tr>
<th>60</th>
<th>T</th>
<th>///</th>
<th>B</th>
<th>184</th>
<th>bit</th>
<th>60</th>
</tr>
</thead>
</table>
```

```plaintext
ex_flag ← 0b0

doi = 0 to 1
  reset_xflags()
  v[0:inf] ← ConvertSWtoFP(VSR[32×BX+B].word[i])
  result.word[i] ← RoundToSP(RN,v)
  if(xx_flag) then SetFX(XX)
  ex_flag ← ex_flag | (XE & xx_flag)
end

eif(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 vector element 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

**VSR Data Layout for VCVXWSP**

<table>
<thead>
<tr>
<th>src = VSR[XB]</th>
</tr>
</thead>
<tbody>
<tr>
<td>SW</td>
</tr>
<tr>
<td>tgt = VSR[XT]</td>
</tr>
<tr>
<td>SP</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>
VSX Vector Convert with round Unsigned Doubleword to Double-Precision format

**XX2-form**

```
xvcvuxddp        XT,XB
```

<table>
<thead>
<tr>
<th></th>
<th>60</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>488</th>
<th>8x</th>
</tr>
</thead>
<tbody>
<tr>
<td>XT</td>
<td>← TX</td>
<td></td>
<td>T</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XB</td>
<td>← BX</td>
<td></td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| ex_flag | ← 0b0

```
do i=0 to 127 by 64
  reset_xflags()
  v{0:inf} ← ConvertUDtoFP(VSR[XB]{i:i+63})
  result{i:i+63} ← RoundToDP(RN,v)
  if(xx_flag) then SetFX(XX)
  ex_flag ← ex_flag | (XE & xx_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

For each vector element 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
```

src = VSR[XB]

<table>
<thead>
<tr>
<th></th>
<th>UD</th>
<th>UD</th>
</tr>
</thead>
</table>

tgt = VSR[XT]

<table>
<thead>
<tr>
<th></th>
<th>DP</th>
<th>DP</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>32</td>
<td>64</td>
</tr>
<tr>
<td>96</td>
<td>127</td>
<td></td>
</tr>
</tbody>
</table>

VSX Vector Convert with round Unsigned Doubleword to Single-Precision format

**XX2-form**

```
xvcvuxdsp        XT,XB
```

<table>
<thead>
<tr>
<th></th>
<th>60</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>424</th>
<th>8x</th>
</tr>
</thead>
<tbody>
<tr>
<td>XT</td>
<td>← TX</td>
<td></td>
<td>T</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XB</td>
<td>← BX</td>
<td></td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| ex_flag | ← 0b0

```
do i=0 to 127 by 64
  reset_xflags()
  v{0:inf} ← ConvertUDtoFP(VSR[XB]{i:i+63})
  result{i+32:i+63} ← RoundToSP(RN,v)
  result{i+64:i+95} ← 0xUUUU_UUUU
  if(xx_flag) then SetFX(XX)
  ex_flag ← ex_flag | (XE & xx_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

For each vector element 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 contents of bits 32:63 of doubleword element i of VSR[XT] are undefined.

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 xvcvuxdsp
```

src = VSR[XB]

<table>
<thead>
<tr>
<th></th>
<th>UD</th>
<th>UD</th>
</tr>
</thead>
</table>

tgt = VSR[XT]

<table>
<thead>
<tr>
<th></th>
<th>SP</th>
<th>undefined</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>32</td>
<td>64</td>
</tr>
<tr>
<td>96</td>
<td>127</td>
<td></td>
</tr>
</tbody>
</table>

**Version 3.0 B**
Chapter 7. Vector-Scalar Floating-Point Operations

### VSX Vector Convert Unsigned Word to Double-Precision format XX2-form

#### xvcvuxwdp

<table>
<thead>
<tr>
<th>60</th>
<th>T</th>
<th>III</th>
<th>B</th>
<th>232</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>6</td>
<td></td>
<td>11</td>
<td>16</td>
</tr>
</tbody>
</table>

\[
\text{do } i = 0 \text{ to } 1 \\
\quad \text{src} \leftarrow \text{bfg\_CONVERT\_FROM\_UI32(VSR[32\times B+8].\text{dword}[i].\text{word}[0])} \\
\quad \text{VSR}[32\times T+T].\text{dword}[i] \leftarrow \text{bfg\_CONVERT\_FROM\_BFP(src)} \\
\text{end}
\]

Let \(XT\) be the value \(32\times T + T\).
Let \(XB\) be the value \(32\times B + B\).

For each vector element \(i\) from 0 to 1, do the following.
Let \(src\) be the unsigned 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 xvcvuxwdp

<table>
<thead>
<tr>
<th>src = VSR[XB]</th>
</tr>
</thead>
<tbody>
<tr>
<td>UW</td>
</tr>
<tr>
<td>tgt = VSR[XT]</td>
</tr>
<tr>
<td>----------------</td>
</tr>
<tr>
<td>DP</td>
</tr>
</tbody>
</table>

### VSX Vector Convert with round Unsigned Word to Single-Precision format XX2-form

#### xvcvuxwsp

<table>
<thead>
<tr>
<th>60</th>
<th>T</th>
<th>III</th>
<th>B</th>
<th>168</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>6</td>
<td></td>
<td>11</td>
<td>16</td>
</tr>
</tbody>
</table>

\[
\text{do } i = 0 \text{ to } 127 \text{ by } 32 \\
\quad \text{reset_xflags||} \\
\quad v[0:inf] \leftarrow \text{ConvertUWFp(VSR[XB][i:i+31])} \\
\quad \text{result}[i:i+31] \leftarrow \text{RoundToSP(RN, v)} \\
\quad \text{if}(\text{xx\_flag}) \text{ then SetFX(XX)} \\
\quad \text{ex\_flag} \leftarrow \text{ex\_flag | (XE \& xx\_flag)} \\
\text{end}\]

\[
\text{if( ex\_flag = 0 ) then VSR[XT] \leftarrow result}
\]

Let \(XT\) be the value \(32\times T + T\).
Let \(XB\) be the value \(32\times B + B\).

For each vector element \(i\) from 0 to 3, do the following.
Let \(src\) be the unsigned integer value 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

#### VSR Data Layout for xvcvuxwsp

<table>
<thead>
<tr>
<th>src = VSR[XB]</th>
</tr>
</thead>
<tbody>
<tr>
<td>UW</td>
</tr>
<tr>
<td>tgt = VSR[XT]</td>
</tr>
<tr>
<td>----------------</td>
</tr>
<tr>
<td>SP</td>
</tr>
</tbody>
</table>
VSX Vector Divide Double-Precision XX3-form

\textbf{xvdivdp} \quad \textbf{XT,XA,XB}

<table>
<thead>
<tr>
<th>0</th>
<th>60</th>
<th>6</th>
<th>A</th>
<th>B</th>
<th>120</th>
<th>UXBM</th>
<th>UXAI</th>
</tr>
</thead>
</table>

\begin{align*}
\text{XT} & \leftarrow 32 \times TX + T \\
\text{XA} & \leftarrow 32 \times AX + A \\
\text{XB} & \leftarrow 32 \times BX + B \\
\text{ex\_flag} & \leftarrow 0b0
\end{align*}

\textup{do } i=0 \text{ to 127 by 64}
\text{reset\_flags()}
\begin{align*}
\text{src1} & \leftarrow \text{VSR}[\text{XA}][i:i+63] \\
\text{src2} & \leftarrow \text{VSR}[\text{XB}][i:i+63] \\
\text{result}[i:i+63] & \leftarrow \text{DivideDP}(\text{src1}, \text{src2}) \\
\text{if}(\text{vxsnan\_flag}) & \text{ then SetFX(VXSNAN)} \\
\text{if}(\text{vxidi\_flag}) & \text{ then SetFX(VXIDI)} \\
\text{if}(\text{vxzdz\_flag}) & \text{ then SetFX(VXZDZ)} \\
\text{if}(\text{ox\_flag}) & \text{ then SetFX(OX)} \\
\text{if}(\text{ux\_flag}) & \text{ then SetFX(UX)} \\
\text{if}(\text{xx\_flag}) & \text{ then SetFX(XX)} \\
\text{if}(\text{zx\_flag}) & \text{ then SetFX(ZX)} \\
\text{ex\_flag} & \leftarrow \text{ex\_flag} | (\text{VE \& vxsnan\_flag}) \\
\text{ex\_flag} & \leftarrow \text{ex\_flag} | (\text{VE \& vxidi\_flag}) \\
\text{ex\_flag} & \leftarrow \text{ex\_flag} | (\text{VE \& vxzdz\_flag}) \\
\text{ex\_flag} & \leftarrow \text{ex\_flag} | (\text{OE \& ox\_flag}) \\
\text{ex\_flag} & \leftarrow \text{ex\_flag} | (\text{OE \& ux\_flag}) \\
\text{ex\_flag} & \leftarrow \text{ex\_flag} | (\text{ZE \& zx\_flag}) \\
\text{ex\_flag} & \leftarrow \text{ex\_flag} | (\text{XE \& xx\_flag}) \\
\end{align*}
\textup{end}

\textup{if}(\text{ex\_flag} = 0) \text{ then VSR[XT] } \leftarrow \text{result}

Let \text{XT} be the value $32 \times TX + T$.  
Let \text{XA} be the value $32 \times AX + A$.  
Let \text{XB} be the value $32 \times BX + B$.

For each vector element \(i\) from 0 to 1, do the following.

Let \text{src1} be the double-precision floating-point operand in doubleword element \(i\) of \text{VSR}[\text{XA}].

Let \text{src2} be the double-precision floating-point operand in doubleword element \(i\) of \text{VSR}[\text{XB}].

\text{src1} is divided\(^1\) by \text{src2}, producing a quotient having unbounded range and precision.

The quotient is normalized\(^2\).

See Table 110.

The intermediate result is rounded to double-precision using the rounding mode specified by \text{RN}.

See Table 50, “Scalar Floating-Point Intermediate Result Handling,” on page 515.

---

\(^1\) 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 110. Actions for xvddivdp (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 vxxi_flag ← 1</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← +Zero</td>
<td>v ← -Zero</td>
<td>v ← +Infinity</td>
<td>v ← dQNaN vxxi_flag ← 1</td>
<td>v ← src2 vxsnan_flag ← 1</td>
</tr>
<tr>
<td>-NZF</td>
<td>v ← +Zero</td>
<td>v ← D(src1,src2) vxxi_flag ← 1</td>
<td>v ← +Infinity vxxi_flag ← 1</td>
<td>v ← -Infinity vxxi_flag ← 1</td>
<td>v ← +Zero</td>
<td>v ← -Zero</td>
<td>v ← +NZF vxxi_flag ← 1</td>
<td>v ← src2 vxsnan_flag ← 1</td>
</tr>
<tr>
<td>-Zero</td>
<td>v ← +Zero</td>
<td>v ← -Infinity vxxi_flag ← 1</td>
<td>v ← -Infinity vxxi_flag ← 1</td>
<td>v ← +Zero</td>
<td>v ← +Zero</td>
<td>v ← -Infinity vxxi_flag ← 1</td>
<td>v ← src2 vxsnan_flag ← 1</td>
<td></td>
</tr>
<tr>
<td>+Zero</td>
<td>v ← -Zero</td>
<td>v ← +Zero</td>
<td>v ← dQNaN vxxi_flag ← 1</td>
<td>v ← -Zero</td>
<td>v ← -Zero</td>
<td>v ← +Zero</td>
<td>v ← +Zero</td>
<td>v ← src2 vxsnan_flag ← 1</td>
</tr>
<tr>
<td>+NZF</td>
<td>v ← -Zero</td>
<td>v ← D(src1,src2) vxxi_flag ← 1</td>
<td>v ← -Infinity vxxi_flag ← 1</td>
<td>v ← +Zero</td>
<td>v ← +Zero</td>
<td>v ← +Zero</td>
<td>v ← +Zero</td>
<td>v ← src2 vxsnan_flag ← 1</td>
</tr>
<tr>
<td>+Infinity</td>
<td>v ← -Infinity vxxi_flag ← 1</td>
<td>v ← +Infinity vxxi_flag ← 1</td>
<td>v ← +Infinity vxxi_flag ← 1</td>
<td>v ← +Infinity vxxi_flag ← 1</td>
<td>v ← +Infinity vxxi_flag ← 1</td>
<td>v ← +Infinity vxxi_flag ← 1</td>
<td>v ← src2 vxsnan_flag ← 1</td>
<td></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 vxsnan_flag ← 1</td>
</tr>
<tr>
<td>SNaN</td>
<td>v ← Q(src1) vxsnan_flag ← 1</td>
<td>v ← Q(src1) vxsnan_flag ← 1</td>
<td>v ← Q(src1) vxsnan_flag ← 1</td>
<td>v ← Q(src1) vxsnan_flag ← 1</td>
<td>v ← Q(src1) vxsnan_flag ← 1</td>
<td>v ← Q(src1) vxsnan_flag ← 1</td>
<td>v ← Q(src1) vxsnan_flag ← 1</td>
<td>v ← Q(src1) 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).
- **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 Vector Divide Single-Precision XX3-form

xvdivsp XT,XA,XB

| 60 | 6 | T | 31 | A | 16 | B | 88 |
|----|---|---|----|---|----|---|----|---|

XT ← TX || T
XA ← AX || A
XB ← BX || B

ex_flag ← 0b0

doi=0 to 327 by 32
reset_xflags()
src1 ← VSR[XA][i:i+31]
src2 ← VSR[XB][i:i+31]
v{i:i+31} ← DivideSP(src1,src2)

result[i:i+31] ← RoundToSP(RN,v)
if(vxsnan_flag) then SetFX(VXSNAN)
if(vxidi_flag) then SetFX(VXIDI)
if(vxisi_flag) then SetFX(VXZDZ)
if(ox_flag) then SetFX(OX)
if(ux_flag) then SetFX(UX)
if(xx_flag) then SetFX(XX)
if(zx_flag) then SetFX(ZX)

ex_flag ← ex_flag | (VE & vxsnan_flag)
ex_flag ← ex_flag | (VE & vxidi_flag)
ex_flag ← ex_flag | (VE & vxisi_flag)
ex_flag ← ex_flag | (OE & ox_flag)
ex_flag ← ex_flag | (UE & ux_flag)
ex_flag ← ex_flag | (ZE & xx_flag)
ex_flag ← ex_flag | (XE & zx_flag)

if(ex_flag = 0) then VSR[XT] ← 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 vector element 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 divided\(^1\) by src2, producing a quotient having unbounded range and precision.

The quotient is normalized\(^2\).

See Table 111.

The intermediate result is rounded to single-precision using the rounding mode specified by RN.

See Table 50, “Scalar Floating-Point Intermediate Result Handling,” on page 515.

\(^1\) 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.
### Chapter 7. Vector-Scalar Floating-Point Operations

#### Table 111. Actions for `xdivsp` (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 vxiidi_flag ← 1</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← dQNaN vxiidi_flag ← 1</td>
<td>v ← src2</td>
<td>v ← Q(src2) vxsnan_flag ← 1</td>
<td></td>
</tr>
<tr>
<td>-NZF</td>
<td>v ← +Zero</td>
<td>v ← D(src1, src2) vixzdz_flag ← 1</td>
<td>v ← -Infinity vixzdz_flag ← 1</td>
<td>v ← dQNaN vixzdz_flag ← 1</td>
<td>v ← D(src1, src2) vixzdz_flag ← 1</td>
<td>v ← src2</td>
<td>v ← Q(src2) vxsnan_flag ← 1</td>
<td></td>
</tr>
<tr>
<td>-Zero</td>
<td>v ← +Zero</td>
<td>v ← dQNaN vixzdz_flag ← 1</td>
<td>v ← dQNaN vixzdz_flag ← 1</td>
<td>v ← -Zero</td>
<td>v ← -Zero</td>
<td>v ← src2</td>
<td>v ← Q(src2) vxsnan_flag ← 1</td>
<td></td>
</tr>
<tr>
<td>+Zero</td>
<td>v ← -Zero</td>
<td>v ← -Infinity vixzdz_flag ← 1</td>
<td>v ← -Infinity vixzdz_flag ← 1</td>
<td>v ← +Zero</td>
<td>v ← +Zero</td>
<td>v ← src2</td>
<td>v ← Q(src2) vxsnan_flag ← 1</td>
<td></td>
</tr>
<tr>
<td>+NZF</td>
<td>v ← -Zero</td>
<td>v ← D(src1, src2) vixzdz_flag ← 1</td>
<td>v ← +Infinity vixzdz_flag ← 1</td>
<td>v ← dQNaN vixzdz_flag ← 1</td>
<td>v ← D(src1, src2) vixzdz_flag ← 1</td>
<td>v ← src2</td>
<td>v ← Q(src2) vxsnan_flag ← 1</td>
<td></td>
</tr>
<tr>
<td>+Infinity</td>
<td>v ← dQNaN vixzdz_flag ← 1</td>
<td>v ← +Infinity vixzdz_flag ← 1</td>
<td>v ← +Infinity vixzdz_flag ← 1</td>
<td>v ← +Infinity vixzdz_flag ← 1</td>
<td>v ← +Infinity vixzdz_flag ← 1</td>
<td>v ← src2</td>
<td>v ← Q(src2) vxsnan_flag ← 1</td>
<td></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) vxsnan_flag ← 1</td>
<td>v ← Q(src1) vxsnan_flag ← 1</td>
<td>v ← Q(src1) vxsnan_flag ← 1</td>
<td>v ← Q(src1) vxsnan_flag ← 1</td>
<td>v ← Q(src1) vxsnan_flag ← 1</td>
<td>v ← Q(src1) vxsnan_flag ← 1</td>
<td>v ← Q(src1) vxsnan_flag ← 1</td>
<td></td>
</tr>
</tbody>
</table>

**Explanation:**

- **src1** The single-precision floating-point value in word element i of VSR[XA] (where i c \{0,1,2,3\}).
- **src2** The single-precision floating-point value in word element i of VSR[XB] (where i c \{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.
VSX Vector Insert Exponent Double-Precision
XX3-form

xviexpdp  XT,XA,XB

<table>
<thead>
<tr>
<th>0</th>
<th>60</th>
<th>T</th>
<th>A</th>
<th>B</th>
<th>248</th>
<th>VSR[XT].dword[1]</th>
</tr>
</thead>
</table>

Let XT be the sum $32 \times TX + T$.
Let XA be the sum $32 \times AX + A$.
Let XB be the sum $32 \times BX + B$.

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[XA].

Let src2 be the unsigned integer value in doubleword element $i$ of VSR[XB].

The contents of bits 0 of src1 are placed into bit 0 of doubleword element $i$ of VSR[XT].

The contents of bits 53:63 of src2 are placed into bits 1:11 of doubleword element $i$ of VSR[XT].

The contents of bits 12:63 of src1 are placed into bits 12:63 of doubleword element $i$ of VSR[XT].

Special Registers Altered:
None

VSR Data Layout for xviexpdp

```
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 Insert Exponent Single-Precision
XX3-form

xviexpsp  XT,XA,XB

<table>
<thead>
<tr>
<th>0</th>
<th>60</th>
<th>T</th>
<th>A</th>
<th>B</th>
<th>216</th>
<th>VSR[XT].word[1]</th>
</tr>
</thead>
</table>

Let XT be the sum $32 \times TX + T$.
Let XA be the sum $32 \times AX + A$.
Let XB be the sum $32 \times BX + B$.

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[XA].

Let src2 be the unsigned integer value in word element $i$ of VSR[XB].

The contents of bits 0 of src1 are placed into bit 0 of word element $i$ of VSR[XT].

The contents of bits 24:31 of src2 are placed into bits 1:8 of word element $i$ of VSR[XT].

The contents of bits 9:31 of src1 are placed into bits 9:31 of word element $i$ of VSR[XT].

Special Registers Altered:
None

VSR Data Layout for xviexpsp

```
```

700  Power ISA™ I
VSX Vector Multiply-Add Double-Precision
XX3-form

\[ \text{xvmaddadp} \quad \text{XT,XA,XB} \]

\[
\begin{array}{cccccc}
60 & 6 & 11 & 16 & 21 & 97 \\
0 & 6 & 10 & 16 & 21 & 98 & 97 \\
\end{array}
\]

\[ \text{xvmaddmdp} \quad \text{XT,XA,XB} \]

\[
\begin{array}{cccccc}
60 & 6 & 11 & 16 & 21 & 105 \\
0 & 6 & 10 & 16 & 21 & 98 & 97 \\
\end{array}
\]

**For xvmaddadp**, do the following.
- Let \( \text{src1} \) be the double-precision floating-point operand in doubleword element \( i \) of VSR[\( \text{XA} \)].
- Let \( \text{src2} \) be the double-precision floating-point operand in doubleword element \( i \) of VSR[\( \text{XB} \)].
- Let \( \text{src3} \) be the double-precision floating-point operand in doubleword element \( i \) of VSR[\( \text{XT} \)].

\( \text{src1} \) is multiplied\(^1\) by \( \text{src3} \), producing a product having unbounded range and precision.

See part 1 of Table 112.

\( \text{src2} \) is added\(^2\) to the product, producing a sum having unbounded range and precision.

The sum is normalized\(^3\).

See part 2 of Table 112.

The intermediate result is rounded to double-precision using the rounding mode specified by \( \text{RN} \).

See Table 50, “Scalar Floating-Point Intermediate Result Handling,” on page 515.

The result is placed into doubleword element \( i \) of VSR[\( \text{XT} \)] in double-precision format.

See Table 98, “Vector Floating-Point Final Result,” on page 661.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[\( \text{XT} \)].

**Special Registers Altered**
- \( \text{FX} \)
- \( \text{OX} \)
- \( \text{UX} \)
- \( \text{XX} \)
- \( \text{VXSNAN} \)
- \( \text{VXSI} \)
- \( \text{VXIMZ} \)

**For xvmaddmdp**, do the following.
- Let \( \text{src1} \) be the double-precision floating-point operand in doubleword element \( i \) of VSR[\( \text{XA} \)].
- Let \( \text{src2} \) be the double-precision floating-point operand in doubleword element \( i \) of VSR[\( \text{XB} \)].
- Let \( \text{src3} \) be the double-precision floating-point operand in doubleword element \( i \) of VSR[\( \text{XT} \)].

Let \( \text{XT} \) be the value \( 32 \times \text{TX} + T \).

Let \( \text{XA} \) be the value \( 32 \times \text{AX} + A \).

Let \( \text{XB} \) be the value \( 32 \times \text{BX} + B \).

For each vector element \( i \) from 0 to 1, do the following.

**For xvmaddadp**, do the following.
- Let \( \text{src1} \) be the double-precision floating-point operand in doubleword element \( i \) of VSR[\( \text{XA} \)].
- Let \( \text{src2} \) be the double-precision floating-point operand in doubleword element \( i \) of VSR[\( \text{XT} \)].
- Let \( \text{src3} \) be the double-precision floating-point operand in doubleword element \( i \) of VSR[\( \text{XB} \)].

Floating-point multiplication is based on exponent addition and multiplication of the significands.

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.

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 xvmadd(a|m)dp

src1 = VSR[XA]

src2 = \texttt{xsmaddap} ? VSR[XT] : VSR[XB]

src3 = \texttt{xsmaddap} ? VSR[XB] : VSR[XT]

tgt = VSR[XT]

0 64 127
### Table 112. Actions for xvmadd(a|m)dp

<table>
<thead>
<tr>
<th>src3</th>
<th>src1</th>
<th>src2</th>
<th>src3</th>
<th>src1</th>
<th>src2</th>
<th>src3</th>
<th>src1</th>
<th>src2</th>
<th>src3</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Infinity</td>
<td>-Infinity</td>
<td>p + ∞</td>
<td>+Infinity</td>
<td>p + ∞</td>
<td>+Infinity</td>
<td>p + ∞</td>
<td>+Infinity</td>
<td>p + ∞</td>
<td>+Infinity</td>
</tr>
<tr>
<td>-Infinity</td>
<td>-Infinity</td>
<td>p + ∞</td>
<td>+Infinity</td>
<td>p + ∞</td>
<td>+Infinity</td>
<td>p + ∞</td>
<td>+Infinity</td>
<td>p + ∞</td>
<td>+Infinity</td>
</tr>
<tr>
<td>-Zero</td>
<td>+Zero</td>
<td>p + 0</td>
<td>+Infinity</td>
<td>p + 0</td>
<td>+Infinity</td>
<td>p + 0</td>
<td>+Infinity</td>
<td>p + 0</td>
<td>+Infinity</td>
</tr>
<tr>
<td>+Zero</td>
<td>-Zero</td>
<td>p + 0</td>
<td>+Infinity</td>
<td>p + 0</td>
<td>+Infinity</td>
<td>p + 0</td>
<td>+Infinity</td>
<td>p + 0</td>
<td>+Infinity</td>
</tr>
<tr>
<td>+Infinity</td>
<td>-Infinity</td>
<td>p + ∞</td>
<td>+Infinity</td>
<td>p + ∞</td>
<td>+Infinity</td>
<td>p + ∞</td>
<td>+Infinity</td>
<td>p + ∞</td>
<td>+Infinity</td>
</tr>
<tr>
<td>QNaN</td>
<td>src1</td>
<td>p + src1</td>
<td>src1</td>
<td>p + src1</td>
<td>src1</td>
<td>p + src1</td>
<td>src1</td>
<td>p + src1</td>
<td>src1</td>
</tr>
<tr>
<td>SNaN</td>
<td>src1</td>
<td>p + src1</td>
<td>src1</td>
<td>p + src1</td>
<td>src1</td>
<td>p + src1</td>
<td>src1</td>
<td>p + src1</td>
<td>src1</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 **xvmaddadp** and **xvmaddmdp**, the double-precision floating-point value in doubleword element i of VSR[XT] (where i ∈ {0, 1}).
- **src3** For **xvmaddadp** and **xvmaddmdp**, 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 the 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.

Chapter 7. Vector-Scalar Floating-Point Operations 703
**VSX Vector Multiply-Add Single-Precision XX3-form**

**xvmaddasp**  
\[ XT,XA,XB \]

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>60</td>
<td>6</td>
<td>A</td>
<td>B</td>
<td>65</td>
<td>VEX/VTX</td>
</tr>
</tbody>
</table>

**xvmaddmsp**  
\[ XT,XA,XB \]

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>60</td>
<td>6</td>
<td>A</td>
<td>B</td>
<td>73</td>
<td>VEX/VTX</td>
</tr>
</tbody>
</table>

\[\begin{align*}
XT & \leftarrow TX || T \\
XA & \leftarrow AX || A \\
XB & \leftarrow BX || B \\
ex_flag & \leftarrow 0b0 \\
do & \text{ i=0 to 127 by 32} \\
\text{reset_xflags()} \\
src1 & \leftarrow VSR[XA]\{i:i+31\} \\
src2 & \leftarrow "xvmaddasp" ? VSR[XT]\{i:i+31\} : VSR[XB]\{i:i+31\} \\
src3 & \leftarrow "xvmaddasp" ? VSR[XB]\{i:i+31\} : VSR[XT]\{i:i+31\} \\
v(\text{inf}) & \leftarrow \text{MultiplyAddSP}(src1,src3,src2) \\
\text{result}\{i:i+63\} & \leftarrow \text{RoundToSP}(RN,v) \\
\text{if}(\text{vsnan_flag}) & \leftarrow \text{SetFX}(VXSNAN) \\
\text{if}(\text{vxiis_flag}) & \leftarrow \text{SetFX}(VXISI) \\
\text{if}(\text{fxnflag}) & \leftarrow \text{SetFX}(FXN) \\
\text{if}(\text{fxflag}) & \leftarrow \text{SetFX}(FX) \\
ex_flag & \leftarrow ex_flag | (VE & vsnan_flag) \\
ex_flag & \leftarrow ex_flag | (VE & vxiis_flag) \\
ex_flag & \leftarrow ex_flag | (OE & ox_flag) \\
ex_flag & \leftarrow ex_flag | (UX & ux_flag) \\
ex_flag & \leftarrow ex_flag | (XE & xx_flag) \\
ex_flag & \leftarrow ex_flag | (VE & ex_flag) \\
\text{end} \\
\text{if}(\text{ex_flag} = 0) \text{ then VSR[XT]} & \leftarrow \text{result} \\
\end{align*}\]

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 vector element \(i\) 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[XA].
- 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 113.

\(src2\) is added\(^2\) to the product, producing a sum having unbounded range and precision.

The sum is normalized\(^3\).

See part 2 of Table 113.

The intermediate result is rounded to single-precision using the rounding mode specified by \(RN\).

See Table 50, “Scalar Floating-Point Intermediate Result Handling,” on page 515.

The result is placed into word element \(i\) of VSR[XT] in single-precision format.

See Table 98, “Vector Floating-Point Final Result,” on page 661.

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 \((G, R, \text{ and } X)\) 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.
Chapter 7. Vector-Scalar Floating-Point Operations

VSR Data Layout for xvmadd(a|m)sp

| src1 = VSR[XA] | SP | SP | SP | SP |
| src2 = xsmaddasp ? VSR[XT] : VSR[XB] | SP | SP | SP | SP |
| src3 = xsmaddasp ? VSR[XB] : VSR[XT] | SP | SP | SP | SP |
| tgt = VSR[XT] | SP | SP | SP | SP |

0 32 64 96 127
Table 113. Actions for `xvmadd(a|m)sp`

<table>
<thead>
<tr>
<th>src1</th>
<th>src2</th>
<th>src3</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\langle \infty, \infty, \infty \rangle$</td>
<td>$\langle \infty, \infty, \infty \rangle$</td>
<td>$\langle dQNaN, v, p \rangle$</td>
</tr>
<tr>
<td>$\langle \infty, \infty, \infty \rangle$</td>
<td>$\langle \infty, \infty, \infty \rangle$</td>
<td>$\langle dQNaN, v, p \rangle$</td>
</tr>
<tr>
<td>$\langle \infty, \infty, \infty \rangle$</td>
<td>$\langle \infty, \infty, \infty \rangle$</td>
<td>$\langle dQNaN, v, p \rangle$</td>
</tr>
<tr>
<td>$\langle \infty, \infty, \infty \rangle$</td>
<td>$\langle \infty, \infty, \infty \rangle$</td>
<td>$\langle dQNaN, v, p \rangle$</td>
</tr>
<tr>
<td>$\langle \infty, \infty, \infty \rangle$</td>
<td>$\langle \infty, \infty, \infty \rangle$</td>
<td>$\langle dQNaN, v, p \rangle$</td>
</tr>
<tr>
<td>$\langle \infty, \infty, \infty \rangle$</td>
<td>$\langle \infty, \infty, \infty \rangle$</td>
<td>$\langle dQNaN, v, p \rangle$</td>
</tr>
<tr>
<td>$\langle \infty, \infty, \infty \rangle$</td>
<td>$\langle \infty, \infty, \infty \rangle$</td>
<td>$\langle dQNaN, v, p \rangle$</td>
</tr>
<tr>
<td>$\langle \infty, \infty, \infty \rangle$</td>
<td>$\langle \infty, \infty, \infty \rangle$</td>
<td>$\langle dQNaN, v, p \rangle$</td>
</tr>
<tr>
<td>$\langle \infty, \infty, \infty \rangle$</td>
<td>$\langle \infty, \infty, \infty \rangle$</td>
<td>$\langle dQNaN, v, p \rangle$</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**: For `xvmaddasp`, the single-precision floating-point value in word element $i$ of VSR[XT] (where $i \in \{0, 1, 2, 3\})$.
- **src3**: For `xvmaddasp`, the single-precision floating-point value in word element $i$ of VSR[XB] (where $i \in \{0, 1, 2, 3\})$.
- **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.
VSX Vector Maximum Double-Precision XX3-form

\[
xvmaxdp \quad XT, XA, XB
\]

\[
\begin{array}{ccccccc}
0 & T & A & B & 224 & X & B
\end{array}
\]

\[
\begin{array}{ccccccc}
0 & TX & \| & T & & & \\
1 & AX & \| & A & & & \\
2 & BX & \| & B & & & \\
\end{array}
\]

\[
ex\text{flag} \leftarrow 0b0
\]

do i = 0 to 127 by 64

\[
\begin{align*}
\text{reset}_x\text{flags}() & \\
\text{src1} & \leftarrow \text{VSR}[XA]\{i:i+63\} \\
\text{src2} & \leftarrow \text{VSR}[XB]\{i:i+63\} \\
\text{result}\{i:i+63\} & \leftarrow \text{MaximumDP}(\text{src1}, \text{src2}) \\
\text{if}(\text{vxsnan}\text{flag}) & \text{then SetFX(VXSNAN)} \\
\text{ex}\text{flag} & \leftarrow \text{ex}\text{flag} \mid (\text{VE} \& \text{vxsnan}\text{flag}) \\
\end{align*}
\]

end

\[
\text{if( ex}\text{flag} = 0 ) \text{then VSR}[XT] \leftarrow \text{result}
\]

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 vector element \(i\) 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 \(\text{VSR}[XB]\).

If \(\text{src1}\) is greater than \(\text{src2}\), \(\text{src1}\) is placed into doubleword element \(i\) of \(\text{VSR}[XT]\) in double-precision format. Otherwise, \(\text{src2}\) is placed into doubleword element \(i\) of \(\text{VSR}[XT]\) 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 114.

If a trap-enabled exception occurs in any element of the vector, no results are written to \(\text{VSR}[XT]\).

Special Registers Altered

\[
\begin{array}{ll}
FX & \text{VXSNAN}
\end{array}
\]
<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(src2)</td>
<td>T(src1)</td>
<td>T(src2)</td>
<td>T(src1)</td>
<td>T(Q(src1))</td>
<td>T(Q(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(Q(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(Q(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(src2)</td>
<td>T(Q(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(Q(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(Q(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 i of VSR[XA] (where i ∈ {0,1}).
src2 The double-precision floating-point value in doubleword element i of VSR[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 (i ∈ {0,1}) of VSR[XT] in double-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). If VE=1, update of VSR[XT] is suppressed.

Table 114. Actions for xvmaxdp
Chapter 7. Vector-Scalar Floating-Point Operations

VSX Vector Maximum Single-Precision XX3-form

\( \text{xvmaxsp} \) \( XT,XA,XB \)

<table>
<thead>
<tr>
<th>60</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>192</th>
</tr>
</thead>
<tbody>
<tr>
<td>XT</td>
<td>←</td>
<td>TX</td>
<td></td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>XA</td>
<td>←</td>
<td>AX</td>
<td></td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>XB</td>
<td>←</td>
<td>BX</td>
<td></td>
<td>B</td>
<td></td>
</tr>
<tr>
<td>ex_flag</td>
<td>←</td>
<td>0b0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

do i=0 to 127 by 32
  reset_xflags()
  src1 ← VSR[XA]{i:i+31}
  src2 ← VSR[XB]{i:i+31}
  result{i:i+63} ← MaximumSP(src1,src2)
  if(vxsnan_flag) then SetFX(VXSNAN)
  ex_flag ← ex_flag | (VE & vxsnan_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result

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 vector element \( i \) 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].

  If \( \text{src1} \) is greater than \( \text{src2} \), \( \text{src1} \) is placed into word element \( i \) of VSR[XT] in single-precision format. Otherwise, \( \text{src2} \) 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 \( \text{VE}=0 \) is that SNaN converted to a QNaN.

  See Table 115.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered

\( \text{FX} \) \( \text{VXSNAN} \)
<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(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>-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(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(src1)</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(src1)</td>
<td>T(src1)</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(src1)</td>
<td>T(Q(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(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(Q(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(src1)</td>
<td>T(Q(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>
</tr>
</tbody>
</table>

**Explanation:**

src1 The single-precision floating-point value in word element i of VSR[XA] (where i \(c\) {0,1,2,3}).
src2 The single-precision floating-point value in word element i of VSR[XT] (where i \(c\) {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.
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). If VE=1, update of VSR[XT] is suppressed.

**Table 115. Actions for xvmaxsp**
**VSX Vector Minimum Double-Precision XX3-form**

`xvmindp  XT,XA,XB`

<table>
<thead>
<tr>
<th>60</th>
<th>6</th>
<th>A</th>
<th>B</th>
<th>232</th>
</tr>
</thead>
<tbody>
<tr>
<td>TX</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td></td>
</tr>
</tbody>
</table>

1. Let `XT` be the value of `32×TX + T`.
2. Let `XA` be the value of `32×AX + A`.
3. Let `XB` be the value of `32×BX + B`.

For each vector element `i` from 0 to 1, do the following.

1. Let `src1` be the double-precision floating-point operand in doubleword element `i` of `VSR[XA]`.
2. Let `src2` be the double-precision floating-point operand in doubleword element `i` of `VSR[XB]`.
3. 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 116.

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 Minimum Double-Precision XX3-form**

`xvmindp  XT,XA,XB`

```plaintext
do i=0 to 127 by 64
reset_xflags();
src1 ← VSR[XX]{i:i+64}
src2 ← VSR[XX]{i:i+64}
result{i:i+64} ← MinimumDP(src1,src2)
if(vxsnan_flag) then SetFX(VXSNAN)
ex_flag ← ex_flag | (VE & vxsnan_flag)
end
if( ex_flag = 0 ) then VSR[XT]
```

See Table 116.
<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(Q(src2))</td>
<td>T(Q(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(src1)</td>
<td>T(Q(src2))</td>
<td>T(Q(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(src1)</td>
<td>T(Q(src2))</td>
<td>T(Q(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(Q(src2))</td>
<td>T(Q(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(Q(src2))</td>
<td>T(Q(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 VSR[XA] (where i ∈ {0,1}).
src2 The double-precision floating-point value in doubleword element i of VSR[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 (i ∈ {0,1}) of VSR[XT] in double-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). If VE=1, update of VSR[XT] is suppressed.

**Table 116. Actions for xvmindp**
VSX Vector Minimum Single-Precision XX3-form

\text{vxminsp} \quad \text{XT,XA,XB}

<table>
<thead>
<tr>
<th></th>
<th>60</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
</tr>
</thead>
<tbody>
<tr>
<td>XT</td>
<td>\leftarrow TX</td>
<td></td>
<td>T</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XA</td>
<td>\leftarrow AX</td>
<td></td>
<td>A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XB</td>
<td>\leftarrow BX</td>
<td></td>
<td>B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ex_flag</td>
<td>\leftarrow 0b0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\text{do } i=0 \text{ to } 127 \text{ by } 32

\text{reset_xflags}();

\text{src1} \leftarrow \text{VSR}[XA]_{i:i+31}; \quad \text{src2} \leftarrow \text{VSR}[XB]_{i:i+31}; \\
\text{result}_{i:i+31} \leftarrow \text{MinimumSP(src1,src2)}; \quad \text{ex_flag} \leftarrow \text{ex_flag} \text{ or } (\text{VE} \text{ and vxsnan_flag}); \\
\text{end}

\text{if( ex_flag } = 0 \text{ ) then VSR}[XT] \leftarrow \text{result}

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 vector element \(i\) 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].

If \(\text{src1}\) is less than \(\text{src2}\), \(\text{src1}\) is placed into word element \(i\) of VSR[XT] in single-precision format. Otherwise, \(\text{src2}\) 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 117.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered

\text{FX} \quad \text{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(src1)</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(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(src1)</td>
<td>T(src1)</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(src1)</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(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[XA] (where i c {0,1,2,3}).
- **src2** The single-precision floating-point value in word element i of VSR[XT] (where i c {0,1,2,3}).
- **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 word element i (i c {0,1,2,3}) of VSR[XT] in single-precision format.
- **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). If VE=1, update of VSR[XT] is suppressed.

**Table 117. Actions for xvminsp**
## VSX Vector Multiply-Subtract
### Double-Precision XX3-form

### xvmsubadp

<table>
<thead>
<tr>
<th>60</th>
<th>T</th>
<th>A</th>
<th>B</th>
<th>113</th>
<th>V/S</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>113</td>
<td>28</td>
</tr>
</tbody>
</table>

### xvmsubmdp

<table>
<thead>
<tr>
<th>60</th>
<th>T</th>
<th>A</th>
<th>B</th>
<th>121</th>
<th>V/S</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>121</td>
<td>28</td>
</tr>
</tbody>
</table>

Let $\text{XT}$ be the value $32 \times \text{T} + \text{T}$.
Let $\text{XA}$ be the value $32 \times \text{A} + \text{A}$.
Let $\text{XB}$ be the value $32 \times \text{B} + \text{B}$.

For each vector element $i$ from 0 to 127, do the following.

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.

For **xvmsubadp**, do the following.

- Let $\text{src1}$ be the double-precision floating-point operand in doubleword element $i$ of VSR[XA].
- Let $\text{src2}$ be the double-precision floating-point operand in doubleword element $i$ of VSR[XB].
- Let $\text{src3}$ be the double-precision floating-point operand in doubleword element $i$ of VSR[XT].

$\text{src1}$ is multiplied\(^1\) by $\text{src3}$, producing a product having unbounded range and precision.

See part 1 of Table 118.

$\text{src2}$ 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 118.

The intermediate result is rounded to double-precision using the rounding mode specified by RN.

See Table 50, “Scalar Floating-Point Intermediate Result Handling,” on page 515.

The result is placed into doubleword element $i$ of VSR[XT] in double-precision format.

See Table 98, “Vector Floating-Point Final Result,” on page 661.

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 xvmsub(a|m)dp

<table>
<thead>
<tr>
<th>src1 = VSR[XA]</th>
</tr>
</thead>
<tbody>
<tr>
<td>DP</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>src2 = <em>xvmsubadp</em>? VSR[XT] : VSR[XB]</th>
</tr>
</thead>
<tbody>
<tr>
<td>DP</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>src3 = <em>xvmsubadp</em>? VSR[XB] : VSR[XB]</th>
</tr>
</thead>
<tbody>
<tr>
<td>DP</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>tgt = VSR[XT]</th>
</tr>
</thead>
<tbody>
<tr>
<td>DP</td>
</tr>
</tbody>
</table>

0 64 127
<table>
<thead>
<tr>
<th>Part 1: Multiply</th>
<th>src3</th>
<th>src1</th>
<th>src2</th>
<th>src3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-Infinity</td>
<td>p + Infinity p + Infinity p + 0NaN vximz_flag ← 1</td>
<td>p + 0NaN vximz_flag ← 1</td>
<td>p ← 0Infinity p ← Infinity p ← src3 p ← Q(src3) vximz_flag ← 1</td>
<td></td>
</tr>
<tr>
<td>-NZF</td>
<td>p + Infinity p + M(src1, src3) p ← +Zero p ← 0Zero p ← M(src1, src3) p ← +Infinity p ← src3 p ← Q(src3) vximz_flag ← 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-Zero</td>
<td>p ← 0NaN vximz_flag ← 1 p ← +Zero p ← 0Zero p ← 0Zero p ← 0NaN vximz_flag ← 1 p ← src3 p ← src3 p ← src3 p ← src3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+Zero</td>
<td>p ← 0NaN vximz_flag ← 1 p ← -Zero p ← +Zero p ← +Zero p ← 0NaN vximz_flag ← 1 p ← src3 p ← src3 p ← src3 p ← src3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+NZF</td>
<td>p ← -Infinity p ← -Zero p ← 0Zero p ← 0Zero p ← 0NaN vximz_flag ← 1 p ← src3 p ← src3 p ← src3 p ← src3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Infinity</td>
<td>p ← +Infinity p ← +Infinity p ← +Infinity p ← +Infinity p ← 0NaN vximz_flag ← 1 p ← src3 p ← src3 p ← src3 p ← src3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>QNaN</td>
<td>p ← src1 p ← src1 p ← src1 p ← src1 p ← src1 p ← src1 p ← src1 p ← src1 p ← src1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SNaN</td>
<td>p ← Q(src1) vximz_flag ← 1 p ← Q(src1) vximz_flag ← 1 p ← Q(src1) vximz_flag ← 1 p ← Q(src1) vximz_flag ← 1 p ← Q(src1) vximz_flag ← 1 p ← Q(src1) vximz_flag ← 1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Part 2: Subtract</th>
<th>src2</th>
<th>src1</th>
<th>src2</th>
<th>src1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-Infinity</td>
<td>v ← 0NaN vximz_flag ← 1 v ← -Infinity v ← -Infinity v ← -Infinity v ← -Infinity v ← src2 v ← Q(src2) vximz_flag ← 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-NZF</td>
<td>v ← +Infinity v ← S(p, src2) v ← p v ← S(p, src2) v ← +Infinity v ← src2 v ← Q(src2) vximz_flag ← 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-Zero</td>
<td>v ← +Infinity v ← src2 v ← Rezd v ← src2 v ← src2 v ← src2 v ← src2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+Zero</td>
<td>v ← +Infinity v ← src2 v ← +Zero v ← src2 v ← src2 v ← src2 v ← src2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+NZF</td>
<td>v ← +Infinity v ← S(p, src2) v ← p v ← S(p, src2) v ← +Infinity v ← src2 v ← Q(src2) vximz_flag ← 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Infinity</td>
<td>v ← +Infinity v ← +Infinity v ← +Infinity v ← +Infinity v ← 0NaN vximz_flag ← 1 v ← src2 v ← Q(src2) vximz_flag ← 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>QNaN &amp; src1 is a NaN</td>
<td>v ← p v ← p v ← p v ← p v ← p v ← p v ← p</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>QNaN &amp; src1 not a NaN</td>
<td>v ← p v ← p v ← p v ← p v ← p v ← p v ← src2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Explanation:**

- **src1** The double-precision floating-point value in doubleword element i of VSR[XA] (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\}).
- **d0NaN** Default quiet NaN (`0x7FFB_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.

Table 118. Actions for `xvmsub(a|m)dp`
VSX Vector Multiply-Subtract Single-Precision
XX3-form

For \texttt{xvmsubasp}, 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].
- Let \(\text{src3}\) be the single-precision floating-point operand in word element \(i\) of VSR[XT].

\(\text{src1}\) is multiplied\(^1\) by \(\text{src3}\), producing a product having unbounded range and precision.

See part 1 of Table 119.

\(\text{src2}\) 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 119.

The intermediate result is rounded to single-precision using the rounding mode specified by \(\text{RN}\).

See Table 50, “Scalar Floating-Point Intermediate Result Handling,” on page 515.

The result is placed into word element \(i\) of VSR[XT] in single-precision format.

See Table 98, “Vector Floating-Point Final Result,” on page 661.

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

For \texttt{xvmsubmsp}, 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].
- Let \(\text{src3}\) be the single-precision floating-point operand in word element \(i\) of VSR[XT].

Let \(\text{XT}\) be the value \(32 \times T_{\text{X}} + T\).
Let \(\text{XA}\) be the value \(32 \times A_{\text{X}} + A\).
Let \(\text{XB}\) be the value \(32 \times B_{\text{X}} + B\).

For each vector element \(i\) from 0 to 3, do the following.

For \texttt{xvmsubasp}, 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].
- Let \(\text{src3}\) be the single-precision floating-point operand in word element \(i\) of VSR[XT].

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 xvmsub(a|m)sp

**src1 = VSR[XA]**

<table>
<thead>
<tr>
<th></th>
<th>SP</th>
<th>SP</th>
<th>SP</th>
<th>SP</th>
</tr>
</thead>
</table>

**src2 = xvmsubasp ? VSR[XT] : VSR[XB]**

<table>
<thead>
<tr>
<th></th>
<th>SP</th>
<th>SP</th>
<th>SP</th>
<th>SP</th>
</tr>
</thead>
</table>

**src3 = xvmsubasp ? VSR[XB] : VSR[XT]**

<table>
<thead>
<tr>
<th></th>
<th>SP</th>
<th>SP</th>
<th>SP</th>
<th>SP</th>
</tr>
</thead>
</table>

**tgt = VSR[XT]**

<table>
<thead>
<tr>
<th></th>
<th>SP</th>
<th>SP</th>
<th>SP</th>
<th>SP</th>
</tr>
</thead>
</table>

0 32 64 96 127
### Table 119. Actions for `xvmsub(a|m)sp`

<table>
<thead>
<tr>
<th>src1</th>
<th>src2</th>
<th>src3</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Infinity</td>
<td>-Infinity</td>
<td>+Zero</td>
</tr>
<tr>
<td>-Infinity</td>
<td>+Zero</td>
<td>-Infinity</td>
</tr>
<tr>
<td>-Infinity</td>
<td>+Infinity</td>
<td>QNaN</td>
</tr>
<tr>
<td>+Zero</td>
<td>+Infinity</td>
<td>SNaN</td>
</tr>
<tr>
<td>+Infinity</td>
<td>+Infinity</td>
<td>QNaN</td>
</tr>
<tr>
<td>QNaN</td>
<td>+Infinity</td>
<td>SNaN</td>
</tr>
<tr>
<td>SNaN</td>
<td>+Infinity</td>
<td>SNaN</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 `xvmsubasp`, the single-precision floating-point value in word element i of VSR[XT] (where i ∈ {0,1,2,3}).
- **src3**: For `xvmsubasp`, 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). 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.
- **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 Double-Precision XX3-form**

### xvmuldp XT,XA,XB

<table>
<thead>
<tr>
<th></th>
<th>T</th>
<th>A</th>
<th>B</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>60</td>
<td>16</td>
<td>21</td>
<td>127</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 \( 32 \times BX + B \).

For each vector element \( i \) 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 120.

The intermediate result is rounded to double-precision using the rounding mode specified by \( RN \).

See Table 50, “Scalar Floating-Point Intermediate Result Handling,” on page 515.

The result is placed into doubleword element \( i \) of VSR[XT] in double-precision format.

---

\(^1\) 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.
## Explanation:

- **src1**: The double-precision floating-point value in doubleword element $i$ of VSR[XA] (where $i \in \{0, 1\}$).
- **src2**: The double-precision floating-point value in doubleword element $i$ of VSR[XB] (where $i \in \{0, 1\}$).
- **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 120. Actions for xmuldp

<table>
<thead>
<tr>
<th>src2</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>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>vximz_flag ← 1</td>
<td>v ← +Zero</td>
<td>v ← +Zero</td>
<td>v ← –Zero</td>
<td>–Zero</td>
<td>v ← src2</td>
<td>v ← Q(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>–Zero</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>–Zero</td>
<td>v ← src2</td>
<td>v ← Q(src2)</td>
</tr>
<tr>
<td>QNaN</td>
<td>v ← src1</td>
<td>vxsnan_flag ← 1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>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>
</tr>
</tbody>
</table>
Chapter 7. Vector-Scalar Floating-Point Operations

VSX Vector Multiply Single-Precision XX3-form

xvmulsp XT,XA,XB

<table>
<thead>
<tr>
<th>60</th>
<th>6</th>
<th>48</th>
<th>16</th>
<th>16</th>
<th>80</th>
</tr>
</thead>
<tbody>
<tr>
<td>XT</td>
<td>← TX</td>
<td></td>
<td>T</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XA</td>
<td>← AX</td>
<td></td>
<td>A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XB</td>
<td>← BX</td>
<td></td>
<td>B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ex_flag</td>
<td>← 0b0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

do i=0 to 127 by 32
reset_xflags();
src1 ← VSR[XA]{i:i+31}
src3 ← VSR[XB]{i:i+31}
v{0:inf} ← MultiplySP(src1, src3)
result{0:inf} ← RoundToSP(RN, v)
if(vxsnan_flag) then SetFX(VXSNAN)
if(vximz_flag) then SetFX(VXIMZ)
if(ox_flag) then SetFX(OX)
if(ux_flag) then SetFX(UX)
if(xx_flag) then SetFX(XX)
ex_flag ← ex_flag | (VE & vxsnan_flag)
ex_flag ← ex_flag | (VE & vximz_flag)
ex_flag ← ex_flag | (OE & ox_flag)
ex_flag ← ex_flag | (UE & ux_flag)
ex_flag ← ex_flag | (XE & xx_flag)
end

if(ex_flag = 0) then VSR[XT] ← 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 vector element 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 121.

The intermediate result is rounded to single-precision using the rounding mode specified by RN.

See Table 50, "Scalar Floating-Point Intermediate Result Handling," on page 515.

The result is placed into word element i of VSR[XT] in single-precision format.

---

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 121. Actions for xvmulsp

<table>
<thead>
<tr>
<th>src2</th>
<th>src1</th>
<th>src2</th>
<th>src2</th>
<th>src2</th>
<th>src2</th>
<th>src2</th>
<th>src2</th>
<th>src2</th>
<th>src2</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>
<td>vxsnan_flag ← 1</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>
<td>vxsnan_flag ← 1</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 ← +Infinity</td>
<td>v ← src2</td>
<td>v ← Q(src2)</td>
<td>vxsnan_flag ← 1</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 ← +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 ← M(src1,src2)</td>
<td>v ← -Zero</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>
<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 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**

\[ \text{xvnabsdp } \text{XT, XB} \]

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>60</td>
<td>T</td>
<td>///</td>
<td>B</td>
<td>489</td>
<td>80</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
\text{XT} & \leftarrow T \| T \\
\text{XB} & \leftarrow B \| B \\
\end{align*}
\]

do \ i = 0 \ to \ 127 \ by \ 64 \\
\text{VSR}[\text{XT}]_{i:i+63} & \leftarrow 0b1 \| \text{VSR}[\text{XB}]_{i+1:i+63} \\
\text{end}

Let \ XT \ be \ the \ value \ 32 \times TX + T. \\
Let \ XB \ be \ the \ value \ 32 \times BX + B.

For each vector element \ i \ from \ 0 \ to \ 1, \ do \ the \ following. 

The contents of doubleword element \ i \ of \ VSR[\text{XB}], with bit 0 set to 1, is placed into doubleword element \ i \ of \ VSR[\text{XT}].

**Special Registers Altered**

None

**VSR Data Layout for xvnabsdp**

\[
\begin{array}{cccc}
\text{src} & = & \text{VSR}[\text{XB}] \\
\text{DP} & \quad & \text{DP} \\
\text{tgt} = & \text{VSR}[\text{XT}] & \text{DP} & \text{DP} \\
0 & \quad & 64 & \quad 127 \\
\end{array}
\]

**VSX Vector Negative Absolute Single-Precision XX2-form**

\[ \text{xvnabssp } \text{XT, XB} \]

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>60</td>
<td>T</td>
<td>///</td>
<td>B</td>
<td>425</td>
<td>80</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
\text{XT} & \leftarrow T \| T \\
\text{XB} & \leftarrow B \| B \\
\end{align*}
\]

do \ i = 0 \ to \ 127 \ by \ 32 \\
\text{VSR}[\text{XT}]_{i:i+31} & \leftarrow 0b1 \| \text{VSR}[\text{XB}]_{i+1:i+31} \\
\text{end}

Let \ XT \ be \ the \ value \ 32 \times TX + T. \\
Let \ XB \ be \ the \ value \ 32 \times BX + B.

For each vector element \ i \ from \ 0 \ to \ 3, \ do \ the following. 

The contents of word element \ i \ of \ VSR[\text{XB}], with bit 0 set to 1, is placed into word element \ i \ of \ VSR[\text{XT}].

**Special Registers Altered**

None

**VSR Data Layout for xvnabssp**

\[
\begin{array}{cccc}
\text{src} & = & \text{VSR}[\text{XB}] \\
\text{SP} & \quad & \text{SP} & \text{SP} & \text{SP} \\
\text{tgt} = & \text{VSR}[\text{XT}] & \text{SP} & \text{SP} & \text{SP} & \text{SP} \\
0 & \quad & 32 & \quad 64 & \quad 96 & \quad 127 \\
\end{array}
\]
**VSX Vector Negate Double-Precision XX2-form**

xvnegdp XT,XB

<table>
<thead>
<tr>
<th>0</th>
<th>60</th>
<th>T</th>
<th>///</th>
<th>B</th>
<th>505</th>
</tr>
</thead>
</table>

\[ XT \leftarrow TX || T \]
\[ XB \leftarrow BX || B \]

\[
do \ i = 0 \ to \ 127 \ by \ 64 \\
\quad \text{VSR}[XT]_{i:i+63} \leftarrow \neg \text{VSR}[XB]_{i} || \neg \text{VSR}[XB]_{i+1:i+63}
\]

Let \( XT \) be the value \( 32 \times TX + T \).
Let \( XB \) be the value \( 32 \times BX + B \).

For each vector element \( i \) 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 = VSR[XB]

tgt = VSR[XT]

| 0 | 64 | DP | \| | DP | 127 |

---

**VSX Vector Negate Single-Precision XX2-form**

xvnegsp XT,XB

<table>
<thead>
<tr>
<th>0</th>
<th>60</th>
<th>T</th>
<th>///</th>
<th>B</th>
<th>441</th>
</tr>
</thead>
</table>

\[ XT \leftarrow TX || T \]
\[ XB \leftarrow BX || B \]

\[
do \ i = 0 \ to \ 127 \ by \ 32 \\
\quad \text{VSR}[XT]_{i:i+31} \leftarrow \neg \text{VSR}[XB]_{i} || \neg \text{VSR}[XB]_{i+1:i+31}
\]

Let \( XT \) be the value \( 32 \times TX + T \).
Let \( XB \) be the value \( 32 \times BX + B \).

For each vector element \( i \) 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 = VSR[XB]

tgt = VSR[XT]

<table>
<thead>
<tr>
<th>0</th>
<th>32</th>
<th>SP</th>
<th>|</th>
<th>SP</th>
<th>127</th>
</tr>
</thead>
</table>
VSX Vector Negative Multiply-Add
Double-Precision XX3-form

For \texttt{xvnaddmp}, do the following.
\begin{itemize}
  \item Let \texttt{src1} be the double-precision floating-point operand in doubleword element \(i\) of VSR[\texttt{XA}].
  \item Let \texttt{src2} be the double-precision floating-point operand in doubleword element \(i\) of VSR[\texttt{XB}].
  \item Let \texttt{src3} be the double-precision floating-point operand in doubleword element \(i\) of VSR[\texttt{XT}].
\end{itemize}

\texttt{src1} is multiplied[1] by \texttt{src3}, producing a product having unbounded range and precision.

See part 1 of Table 122.

\texttt{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 122.

The intermediate result is rounded to double-precision using the rounding mode specified by \texttt{RN}.

See Table 50, “Scalar Floating-Point Intermediate Result Handling,” on page 515.

The result is negated and placed into doubleword element \(i\) of VSR[\texttt{XT}] in double-precision format.

See Table 123, “Vector Floating-Point Final Result with Negation,” on page 730.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[\texttt{XT}].

**Special Registers Altered**

\texttt{FX OX UX XX VXSNAN VXISI VXIMZ}

\begin{itemize}
  \item Floating-point multiplication is based on exponent addition and multiplication of the significands.
  \item 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.
  \item 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.
\end{itemize}
VSR Data Layout for xvnadd(a|m)dp

src1 = VSR[XA]

<table>
<thead>
<tr>
<th></th>
<th>DP</th>
<th></th>
</tr>
</thead>
</table>

src2 = \texttt{xsmaddadp} ? VSR[XT] : VSR[XB]

<table>
<thead>
<tr>
<th></th>
<th>DP</th>
<th></th>
</tr>
</thead>
</table>

src3 = \texttt{xsmaddadp} ? VSR[XB] : VSR[XT]

<table>
<thead>
<tr>
<th></th>
<th>DP</th>
<th></th>
</tr>
</thead>
</table>

tgt = VSR[XT]

<table>
<thead>
<tr>
<th></th>
<th>DP</th>
<th></th>
</tr>
</thead>
</table>

0 64 127
Table 122. Actions for `xvmadd(a|m)dp`

**Explanation:**

- **src1** The double-precision floating-point value in doubleword element i of VSR[XA] (where i ∈ {0, 1}).
- **src2** For `xvmaddadp`, the double-precision floating-point value in doubleword element i of VSR[XT] (where i ∈ {0, 1}).
- **src3** For `xvmaddadp`, 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.
- **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 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>
<thead>
<tr>
<th>Part 1: Multiply</th>
<th>src3</th>
<th>src2</th>
<th>src1</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Infinity</td>
<td>v ← -Infinity p ← -Infinity</td>
<td>p ← -Infinity</td>
<td>p ← -Infinity</td>
</tr>
<tr>
<td>-NZF</td>
<td>v ← -Infinity p ← M(src1,src3)</td>
<td>p ← src1</td>
<td>p ← M(src1,src3)</td>
</tr>
<tr>
<td>-Zero</td>
<td>v ← dQNaN vximz_flag ← 1</td>
<td>p ← +Zero</td>
<td>p ← +Zero</td>
</tr>
<tr>
<td>+Zero</td>
<td>v ← -Zero</td>
<td>p ← -Zero</td>
<td>p ← +Zero</td>
</tr>
<tr>
<td>+NZF</td>
<td>v ← dQNaN vximz_flag ← 1</td>
<td>p ← +Zero</td>
<td>p ← +Zero</td>
</tr>
<tr>
<td>QNaN</td>
<td>v ← src1</td>
<td>p ← src1</td>
<td>p ← src1</td>
</tr>
<tr>
<td>SNaN</td>
<td>v ← Q(src1) vximz_flag ← 1</td>
<td>p ← Q(src1) vximz_flag ← 1</td>
<td>p ← Q(src1) vximz_flag ← 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Part 2: Add</th>
<th>src2</th>
<th>src1</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Infinity</td>
<td>v ← -Infinity p ← -Infinity</td>
<td>v ← -Infinity</td>
</tr>
<tr>
<td>-NZF</td>
<td>v ← -Infinity p ← A(p,src2)</td>
<td>p ← p</td>
</tr>
<tr>
<td>-Zero</td>
<td>v ← -Infinity p ← src2</td>
<td>v ← Rezd</td>
</tr>
<tr>
<td>+Zero</td>
<td>v ← -Infinity p ← src2</td>
<td>v ← +Zero</td>
</tr>
<tr>
<td>+NZF</td>
<td>v ← -Infinity p ← A(p,src2)</td>
<td>p ← p</td>
</tr>
<tr>
<td>+Infinity</td>
<td>v ← dQNaN vximz_flag ← 1</td>
<td>v ← +Infinity</td>
</tr>
<tr>
<td>QNaN &amp; src1 is a NaN</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>
</tr>
</tbody>
</table>

| v ← p |
| v ← p |
| v ← p |
| v ← p |
| v ← p |
| v ← p |
| v ← p |
| v ← src2 |
| v ← Q(src2) vximz_flag ← 1 |

**Chapter 7. Vector-Scalar Floating-Point Operations**
### Table 123. Vector Floating-Point Final Result with Negation

<table>
<thead>
<tr>
<th>Case</th>
<th>VE</th>
<th>OE</th>
<th>UE</th>
<th>ZE</th>
<th>XE</th>
<th>vxsnan_flag</th>
<th>vximz_flag</th>
<th>vxisi_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>(T(N(i))                )</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>(T(V); n(VXISI))</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>(T(V); n(VXIMZ))</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>0</td>
<td></td>
<td>(T(V); n(VXSNAN))</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>fx(VXISI), error()</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>1</td>
<td></td>
<td>fx(VXIMZ), error()</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>0</td>
<td></td>
<td>fx(VXSNAN), error()</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td></td>
<td>fx(VXSNAN), fx(VXIMZ), error()</td>
</tr>
<tr>
<td>Normal</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(T(N(i))                )</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(T(N(i)); n(OX))</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td></td>
<td>(T(N(i)); n(VXIMZ))</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>yes</td>
<td>(T(N(i)); n(VXISI), error())</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>0</td>
<td></td>
<td>(T(V); n(VXISI), error())</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>0</td>
<td>yes</td>
<td>(T(V); n(VXIMZ), error())</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>fx(OX), fx(VXISI), error()</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>0</td>
<td></td>
<td>fx(OX), fx(VXISI), error()</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>0</td>
<td>yes</td>
<td>fx(OX), fx(VXISI), error()</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>0</td>
<td>yes</td>
<td>fx(OX), fx(VXISI), error()</td>
</tr>
<tr>
<td>Overflow</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(T(N(i))                )</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(T(N(i)); n(OX), n(VXIMZ))</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td></td>
<td>(T(N(i)); n(OX), n(VXISI), error())</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>no</td>
<td>n(OX), fx(VXISI), error()</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>yes</td>
<td>fx(OX), fx(VXISI), error()</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>yes</td>
<td>fx(OX), fx(VXISI), error()</td>
</tr>
</tbody>
</table>

**Explanation:**

- The results do not depend on this condition.

- `fx(x)` FX is set to 1 if x=0, x is set to 1.

- q The value defined in Table 50, “Scalar Floating-Point Intermediate Result Handling,” on page 515, significand rounded to the target precision, unbounded exponent range.

- r The value defined in Table 50, “Scalar Floating-Point Intermediate Result Handling,” on page 515, 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.

- FI Floating-Point Fraction Inexact status flag, FPSCRFI. This status flag is nonsticky.

- FR Floating-Point Fraction Rounded status flag, FPSCRFR.

- OX Floating-Point Overflow Exception status flag, FPSCRUX.

- error() The system error handler is invoked for the trap-enabled exception if the FE0 and FE1 bits in the Machine State Register are set to any mode other than the ignore-exception mode. Update of the target VSR is suppressed for all vector elements.

- N(x) The value x is 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 ∈ {0, 1} for results with 64-bit elements, and i ∈ {0, 1, 3, 4}) for results with 32-bit elements).

- UX Floating-Point Underflow Exception status flag, FPSCRUX.

- VXSNAN Floating-Point Invalid Operation Exception (SNAN) status flag, FPSCRVXSNAN.

- VXIMZ Floating-Point Invalid Operation Exception (Infinity × Zero) status flag, FPSCRVXIMZ.

- VXISI Floating-Point Invalid Operation Exception (Infinity – Infinity) status flag, FPSCRVXISI.

- XX Floating-Point Inexact Exception status flag, FPSCRXX. The flag is a sticky version of FPSCRxx. When FPSCRxx is set to a new value, the new value of FPSCRxx is set to the result of ORing the old value of FPSCRXX with the new value of FPSCRxx.
### Table 123. Vector Floating-Point Final Result with Negation (Continued)

<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>
</tr>
</thead>
<tbody>
<tr>
<td>Tiny</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>no</td>
<td>no</td>
<td>-</td>
<td>-</td>
<td>T(N(:))</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>yes</td>
<td>no</td>
<td>-</td>
<td>-</td>
<td>T(N(:)), fx(UX), fx(XX)</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>yes</td>
<td>yes</td>
<td>-</td>
<td>-</td>
<td>T(N(:)), fx(UX), fx(XX)</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>no</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>T(N(:)), fx(UX), fx(XX), error()</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>yes</td>
<td>yes</td>
<td>-</td>
<td>-</td>
<td>T(N(:)), fx(UX), fx(XX), error()</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>yes</td>
<td>no</td>
<td>-</td>
<td>-</td>
<td>fx(UX), error()</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>yes</td>
<td>yes</td>
<td>-</td>
<td>-</td>
<td>fx(UX), error()</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>yes</td>
<td>yes</td>
<td>-</td>
<td>-</td>
<td>fx(UX), error()</td>
</tr>
</tbody>
</table>

**Explanation:**

- The results do not depend on this condition.
- \( \text{fx}(x) \) FX is set to 1 if \( x=0 \). \( x \) is set to 1.
- \( q \) The value defined in Table 50, “Scalar Floating-Point Intermediate Result Handling,” on page 515, significand rounded to the target precision, unbounded exponent range.
- \( r \) The value defined in Table 50, “Scalar Floating-Point Intermediate Result Handling,” on page 515, 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.
- \( \text{Fi} \) Floating-Point Fraction Inexact status flag, FPSCR\text{Fi}. This status flag is nonsticky.
- \( \text{FR} \) Floating-Point Fraction Rounded status flag, FPSCR\text{FR}.
- \( \text{OX} \) Floating-Point Overflow Exception status flag, FPSCR\text{OX}.
- \( \text{error}() \) The system error handler is invoked for the trap-enabled exception if the FE0 and FE1 bits in the Machine State Register are set to any mode other than the ignore-exception mode. Update of the target VSR is suppressed for all vector elements.
- \( N(x) \) The value \( x \) is 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 \( i \in \{0,1\} \) for results with 64-bit elements, and \( i \in \{0,1,3,4\} \) for results with 32-bit elements).
- \( \text{UX} \) Floating-Point Underflow Exception status flag, FPSCR\text{UX}.
- \( \text{VXSNAN} \) Floating-Point Invalid Operation Exception (SNaN) status flag, FPSCR\text{VXSNAN}.
- \( \text{VXIMZ} \) Floating-Point Invalid Operation Exception (Infinity × Zero) status flag, FPSCR\text{VXIMZ}.
- \( \text{VXISI} \) Floating-Point Invalid Operation Exception (Infinity – Infinity) status flag, FPSCR\text{VXISI}.
- \( \text{XX} \) Floating-Point Inexact Exception status flag, FPSCR\text{XX}. The flag is a sticky version of FPSCR\text{FX}. When FPSCR\text{FX} is set to a new value, the new value of FPSCR\text{XX} is set to the result of ORing the old value of FPSCR\text{XX} with the new value of FPSCR\text{FX}.
VSX Vector Negative Multiply-Add
Single-Precision XX3-form

For \texttt{xvmaddasp}, do the following.

- Let src1 be the single-precision floating-point operand in word element i of VSR[\texttt{XA}].
- Let src2 be the single-precision floating-point operand in word element i of VSR[\texttt{XT}].
- Let src3 be the single-precision floating-point operand in word element i of VSR[\texttt{XB}].

\texttt{src1} is multiplied\(^1\) by \texttt{src3}, producing a product having unbounded range and precision.

See part 1 of Table 124.

\texttt{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 124.

The intermediate result is rounded to single-precision using the rounding mode specified by \texttt{RN}.

See Table 50, “Scalar Floating-Point Intermediate Result Handling,” on page 515.

The result is negated and placed into word element i of VSR[\texttt{XT}] in single-precision format.

See Table 123, “Vector Floating-Point Final Result with Negation,” on page 730.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[\texttt{XT}].

Special Registers Altered
\texttt{FX} O\texttt{X} U\texttt{X} XX \texttt{VXSNAN} VXISI VXIMZ

\texttt{src1} is multiplied\(^1\) by \texttt{src3}, producing a product having unbounded range and precision.

See part 1 of Table 124.

\texttt{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 124.

The intermediate result is rounded to single-precision using the rounding mode specified by \texttt{RN}.

See Table 50, “Scalar Floating-Point Intermediate Result Handling,” on page 515.

The result is negated and placed into word element i of VSR[\texttt{XT}] in single-precision format.

See Table 123, “Vector Floating-Point Final Result with Negation,” on page 730.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[\texttt{XT}].

Special Registers Altered
\texttt{FX} O\texttt{X} U\texttt{X} XX \texttt{VXSNAN} VXISI VXIMZ

\texttt{src1} is multiplied\(^1\) by \texttt{src3}, producing a product having unbounded range and precision.

For part 124.

The intermediate result is rounded to single-precision using the rounding mode specified by \texttt{RN}.

The result is negated and placed into word element i of VSR[\texttt{XT}] in single-precision format.

See Table 50, “Scalar Floating-Point Intermediate Result Handling,” on page 515.

The result is negated and placed into word element i of VSR[\texttt{XT}] in single-precision format.

See Table 123, “Vector Floating-Point Final Result with Negation,” on page 730.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[\texttt{XT}].

Special Registers Altered
\texttt{FX} O\texttt{X} U\texttt{X} XX \texttt{VXSNAN} VXISI VXIMZ

For \texttt{xvmaddmsp}, do the following.

- Let \texttt{src1} be the single-precision floating-point operand in word element i of VSR[\texttt{XA}].
- Let \texttt{src2} be the single-precision floating-point operand in word element i of VSR[\texttt{XB}].
- Let \texttt{src3} be the single-precision floating-point operand in word element i of VSR[\texttt{XT}].

\texttt{src1} is multiplied\(^1\) by \texttt{src3}, producing a product having unbounded range and precision.

See part 1 of Table 124.

\texttt{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 124.

The intermediate result is rounded to single-precision using the rounding mode specified by \texttt{RN}.

The result is negated and placed into word element i of VSR[\texttt{XT}] in single-precision format.

See Table 50, “Scalar Floating-Point Intermediate Result Handling,” on page 515.

The result is negated and placed into word element i of VSR[\texttt{XT}] in single-precision format.

See Table 123, “Vector Floating-Point Final Result with Negation,” on page 730.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[\texttt{XT}].

Special Registers Altered
\texttt{FX} O\texttt{X} U\texttt{X} XX \texttt{VXSNAN} VXISI VXIMZ

For \texttt{xvmaddmsp}, do the following.

- Let \texttt{src1} be the single-precision floating-point operand in word element i of VSR[\texttt{XA}].
- Let \texttt{src2} be the single-precision floating-point operand in word element i of VSR[\texttt{XB}].
- Let \texttt{src3} be the single-precision floating-point operand in word element i of VSR[\texttt{XT}].

\texttt{src1} is multiplied\(^1\) by \texttt{src3}, producing a product having unbounded range and precision.

See part 1 of Table 124.

\texttt{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 124.

The intermediate result is rounded to single-precision using the rounding mode specified by \texttt{RN}.

The result is negated and placed into word element i of VSR[\texttt{XT}] in single-precision format.

See Table 50, “Scalar Floating-Point Intermediate Result Handling,” on page 515.

The result is negated and placed into word element i of VSR[\texttt{XT}] in single-precision format.

See Table 123, “Vector Floating-Point Final Result with Negation,” on page 730.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[\texttt{XT}].

Special Registers Altered
\texttt{FX} O\texttt{X} U\texttt{X} XX \texttt{VXSNAN} VXISI VXIMZ

\texttt{src1} is multiplied\(^1\) by \texttt{src3}, producing a product having unbounded range and precision.

For part 124.

The intermediate result is rounded to single-precision using the rounding mode specified by \texttt{RN}.

The result is negated and placed into word element i of VSR[\texttt{XT}] in single-precision format.

See Table 50, “Scalar Floating-Point Intermediate Result Handling,” on page 515.

The result is negated and placed into word element i of VSR[\texttt{XT}] in single-precision format.

See Table 123, “Vector Floating-Point Final Result with Negation,” on page 730.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[\texttt{XT}].

Special Registers Altered
\texttt{FX} O\texttt{X} U\texttt{X} XX \texttt{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 `xvnmmadd(a|m)sp`

<table>
<thead>
<tr>
<th>src1 = VSR[XA]</th>
<th>SP</th>
<th>SP</th>
<th>SP</th>
<th>SP</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>src2 = <code>xsmaddadp</code> ? VSR[XT] : VSR[XB]</th>
<th>SP</th>
<th>SP</th>
<th>SP</th>
<th>SP</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>src3 = <code>xsmaddadp</code> ? VSR[XB] : VSR[XT]</th>
<th>SP</th>
<th>SP</th>
<th>SP</th>
<th>SP</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>tgt = VSR[XT]</th>
<th>SP</th>
<th>SP</th>
<th>SP</th>
<th>SP</th>
</tr>
</thead>
</table>

0 32 64 96 127
Table 124. Actions for xvnmadd(a|m)sp

<table>
<thead>
<tr>
<th>Part 1: Multiply</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)</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 ← –Infinity</td>
<td>p ← –Infinity</td>
<td>p ← 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 ← –Infinity</td>
<td>p ← –Infinity</td>
<td>p ← src3</td>
<td>p ← Q(src3)</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)</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>

<table>
<thead>
<tr>
<th>Part 2: Add</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 ← A(p,src2)</td>
<td>v ← p</td>
<td>v ← A(p,src2)</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 ← src2</td>
<td>v ← –Zero</td>
<td>v ← Rezd</td>
<td>v ← src2</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
</tr>
<tr>
<td>+Zero</td>
<td>v ← –Infinity</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 ← –Infinity</td>
<td>v ← A(p,src2)</td>
<td>v ← p</td>
<td>v ← A(p,src2)</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</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 ← +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 xvnmaddasp, the single-precision floating-point value in word element i of VSR[XT] (where i ∈ {0,1,2,3}).
- **src3**: For xvnmaddasp, 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). 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 Vector Negative Multiply-Subtract
Double-Precision XX3-form

For \textit{xvnmsubadp}, do the following.
- Let \( \text{src1} \) be the double-precision floating-point operand in doubleword element \( i \) of \( \text{VSR}[\text{XA}] \).
- Let \( \text{src2} \) be the double-precision floating-point operand in doubleword element \( i \) of \( \text{VSR}[\text{XB}] \).
- Let \( \text{src3} \) be the double-precision floating-point operand in doubleword element \( i \) of \( \text{VSR}[\text{XT}] \).

\( \text{src1} \) is multiplied\(^1\) by \( \text{src3} \), producing a product having unbounded range and precision.

See part 1 of Table 125.

\( \text{src2} \) 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 125.

The intermediate result is rounded to double-precision using the rounding mode specified by \( \text{RN} \).

See Table 50, “Scalar Floating-Point Intermediate Result Handling,” on page 515.

The result is negated and placed into doubleword element \( i \) of \( \text{VSR}[\text{XT}] \) in double-precision format.

See Table 123, “Vector Floating-Point Final Result with Negation,” on page 730.

If a trap-enabled exception occurs in any element of the vector, no results are written to \( \text{VSR}[\text{XT}] \).

\noindent \textbf{Special Registers Altered}\n\begin{itemize}
  \item FX
  \item OX
  \item UX
  \item XX
  \item VXSNAN
  \item VXISI
  \item 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 xvnmsub(a|m)dp

\[ \text{src1} = \text{VSR}[XA] \]

\[
\begin{array}{cc}
\text{DP} & \text{DP} \\
\end{array}
\]

\[ \text{src2} = \text{xvnmsubadp} \ ? \ \text{VSR}[XT] : \text{VSR}[XB] \]

\[
\begin{array}{cc}
\text{DP} & \text{DP} \\
\end{array}
\]

\[ \text{src3} = \text{xvnmsubadp} \ ? \ \text{VSR}[XB] : \text{VSR}[XB] \]

\[
\begin{array}{cc}
\text{DP} & \text{DP} \\
\end{array}
\]

\[ \text{tgt} = \text{VSR}[XT] \]

\[
\begin{array}{ccc}
\text{DP} & 64 & 127 \\
\end{array}
\]
### Table 125. Actions for `xvnmsub(a|m)dp`

#### Part 1: Multiply

<table>
<thead>
<tr>
<th><code>src1</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 ← +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 ← +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>SNaN</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>
</tbody>
</table>

#### Part 2: Subtract

<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 ← +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; <code>src1</code> is a NaN</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; <code>src1</code> not a NaN</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>
</tbody>
</table>

#### Explanation:

- **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 j of VSR[XB] (where j ∈ {0, 1}).
- **src3** For `xvnmsubadp`, the double-precision floating-point value in doubleword element k of VSR[XB] (where k ∈ {0, 1}).
- **dQNaN** Default quiet NaN (Qx7FFB_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.
- **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 Single-Precision XX3-form**

For `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[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 126.

`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 126.

The intermediate result is rounded to single-precision using the rounding mode specified by RN.

See Table 50, “Scalar Floating-Point Intermediate Result Handling,” on page 515.

The result is negated and placed into word element `i` of VSR[XT] in single-precision format.

See Table 123, “Vector Floating-Point Final Result with Negation,” on page 730.

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>VXSI</th>
<th>VXIMZ</th>
</tr>
</thead>
</table>

For `xvnmsubmsp`, 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 `src3` be the single-precision floating-point operand in word element `i` of VSR[XT].

`src1` 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 126.

The intermediate result is rounded to single-precision using the rounding mode specified by RN.

See Table 50, “Scalar Floating-Point Intermediate Result Handling,” on page 515.

The result is negated and placed into word element `i` of VSR[XT] in single-precision format.

See Table 123, “Vector Floating-Point Final Result with Negation,” on page 730.

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>VXSI</th>
<th>VXIMZ</th>
</tr>
</thead>
</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.
VSR Data Layout for xvnmsub(a|m)sp

src1 = VSR[XA]

| SP | SP | SP | SP |

src2 = \texttt{xvnmsubasp} ? VSR[XT] : VSR[XB]

| SP | SP | SP | SP |

src3 = \texttt{xvnmsubasp} ? VSR[XB] : VSR[XT]

| SP | SP | SP | SP |

tgt = VSR[XT]

| SP | SP | SP | SP |

| 0  | 32 | 64 | 96 | 127 |
Table 126. Actions for xvnmsub(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 ← dQNaN</td>
<td>p ← –Infinity</td>
<td>p ← –Infinity</td>
<td>p ← src1</td>
<td>p ← src1</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 ← src1</td>
<td>p ← src1</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 ← src1</td>
<td>p ← src1</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 ← 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)</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>

<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 ← src2</td>
<td>v ← src2</td>
</tr>
<tr>
<td>–NZF</td>
<td>v ← +Infinity</td>
<td>v ← –src2</td>
<td>v ← p</td>
<td>v ← p</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 ← –src2</td>
<td>v ← +Infinity</td>
<td>v ← –src2</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 ← –src2</td>
<td>v ← +Infinity</td>
<td>v ← –src2</td>
<td>v ← src2</td>
<td>v ← src2</td>
</tr>
<tr>
<td>+NZF</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 ← src2</td>
<td>v ← src2</td>
</tr>
<tr>
<td>+Infinity</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 ← src2</td>
<td>v ← src2</td>
</tr>
<tr>
<td>QNaN &amp; src1 is a NaN</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>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 c {0,1,2,3}). |
| src2 | The single-precision floating-point value in word element i of VSR[XT] (where i c {0,1,2,3}). |
| src3 | The single-precision floating-point value in word element i of VSR[XB] (where i c {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. |
| 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}
  60  T  ///  B  201  BXT
  60  6  11  16  21  30  31

XT ← TX || T
XB ← BX || B
ex_flag ← 0b0

\textbf{do i=0 to 127 by 64}
 reset_xflags()
 result\{i:i+63\} ← RoundToIntegerNearAway(VSR[XB]\{i:i+63\})
 if(vsnan_flag) then SetFX(VXSNAN)
 ex_flag ← ex_flag | (VE & vsnan_flag)
 end

\textbf{if( ex_flag = 0 ) then VSR[XT] ← result}
\end{verbatim}

Let XT be the value \(32 \times TX + T\).
Let XB be the value \(32 \times BX + B\).

For each vector element 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 to Nearest Away.

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}

\begin{itemize}
  \item FX
  \item VXSNAN
\end{itemize}

\textbf{VSR Data Layout for xvrdpi}

\begin{itemize}
  \item src = VSR[XB]
  \item tgt = VSR[XT]
\end{itemize}

\begin{verbatim}
\begin{array}{llllll}
  0 & 64 & 127 \\
\end{array}
\end{verbatim}

\hspace{1cm}DP \hspace{1cm}DP

\hspace{1cm}DP \hspace{1cm}DP

VSX Vector Round to Double-Precision Integer Exact using Current rounding mode
XX2-form

\texttt{xvrdpic XT,XB}

\begin{verbatim}
  60  T  ///  B  235  BXT
  60  6  11  16  21  30  31

XT ← TX || T
XB ← BX || B
ex_flag ← 0b0

\textbf{do i=0 to 127 by 64}
 reset_xflags()
 src\{i:i+63\} ← VSR[XB]\{i:i+63\}
 if(RN=0b00) then result\{i:i+63\} ← RoundToIntegerNearEven(src)
 if(RN=0b01) then result\{i:i+63\} ← RoundToIntegerTrunc(src)
 if(RN=0b10) then result\{i:i+63\} ← RoundToIntegerCeil(src)
 if(RN=0b11) then result\{i:i+63\} ← RoundToIntegerFloor(src)
 if(xx_flag) then SetFX(XX)
 end

\textbf{if( ex_flag = 0 ) then VSR[XT] ← result}
\end{verbatim}

Let XT be the value \(32 \times TX + T\).
Let XB be the value \(32 \times BX + B\).

For each vector element 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].

\textbf{Special Registers Altered}

\begin{itemize}
  \item FX
  \item XX
  \item VXSNAN
\end{itemize}

\textbf{VSR Data Layout for xvrdpic}

\begin{itemize}
  \item src = VSR[XB]
  \item tgt = VSR[XT]
\end{itemize}

\begin{verbatim}
\begin{array}{llllll}
  0 & 64 & 127 \\
\end{array}
\end{verbatim}

\hspace{1cm}DP \hspace{1cm}DP

\hspace{1cm}DP \hspace{1cm}DP
VSX Vector Round to Double-Precision Integer using round toward -Infinity XX2-form

\textbf{xvrdpim} \ XT,\ XB

\begin{align*}
\text{src} &= \text{VSR}[\text{XB}] \\
\text{tgt} &= \text{VSR}[\text{XT}] \\
\end{align*}

If a trap-enabled exception occurs in any element of the vector, no results are written to \text{VSR}[\text{XT}].

\textbf{Special Registers Altered}

\begin{itemize}
\item \text{FX VXSNAN}
\end{itemize}

\textbf{VSR Data Layout for xvrdpim}

\begin{tabular}{|c|c|c|c|c|}
\hline
0 & 6 & T & /// & B & 249 \\
\hline
\end{tabular}

\begin{tabular}{|c|c|c|c|c|}
\hline
0 & 6 & T & /// & B & 233 \\
\hline
\end{tabular}

\textbf{xvrdpip} \ XT,\ XB

\begin{align*}
\text{src} &= \text{VSR}[\text{XB}] \\
\text{tgt} &= \text{VSR}[\text{XT}] \\
\end{align*}

If a trap-enabled exception occurs in any element of the vector, no results are written to \text{VSR}[\text{XT}].

\textbf{Special Registers Altered}

\begin{itemize}
\item \text{FX VXSNAN}
\end{itemize}

\textbf{VSR Data Layout for xvrdpip}

\begin{tabular}{|c|c|c|c|c|}
\hline
0 & 6 & T & /// & B & 249 \\
\hline
\end{tabular}

\begin{tabular}{|c|c|c|c|c|}
\hline
0 & 6 & T & /// & B & 233 \\
\hline
\end{tabular}
VSX Vector Round to Double-Precision Integer using round toward Zero XX2-form

xvrdpiz XT XB

Let XT be the value \( 32 \times TX + T \).
Let XB be the value \( 32 \times BX + B \).

For each vector element \( i \) 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

VSR Data Layout for xvrdpiz

<table>
<thead>
<tr>
<th>src = VSR[XB]</th>
<th>tgt = VSR[XT]</th>
</tr>
</thead>
<tbody>
<tr>
<td>DP</td>
<td>DP</td>
</tr>
<tr>
<td>0</td>
<td>64</td>
</tr>
<tr>
<td>64</td>
<td>127</td>
</tr>
</tbody>
</table>
### VSX Vector Reciprocal Estimate

#### Double-Precision XX2-form

**xvredp**  

|  | 60 | T |  /// | B | 218 |  ||  |
|---|---|---|------|---|-----|---|---|
| XT | ← T || T |
| XB | ← B || B |
| ex_flag | ← 0b0 |

**Source Value**  

<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>QNaN2</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 = VSR[XB]</th>
</tr>
</thead>
<tbody>
<tr>
<td>tgt = VSR[XT]</td>
</tr>
</tbody>
</table>

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>64</td>
<td>127</td>
</tr>
</tbody>
</table>

**Operation with various special values of the operand is summarized below.**

![Diagram of VSR Data Layout for xvredp]

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,

\[
\left| \frac{\text{estimate} - \frac{1}{\text{src}}}{\frac{1}{\text{src}}} \right| \leq \frac{1}{16384}
\]

Operation with various special values of the operand is summarized below.
VSX Vector Reciprocal Estimate
Single-Precision XX2-form

xvresp XT, XB

\[
\begin{array}{cccccc}
0 & 60 & 6 & // & 154 & 116 \\
X_T & - & TX || T \\
X_B & - & BX || B \\
ex_flag & - & 000 \\
\end{array}
\]

do i = 0 to 3
    reset_xflags() \\
    v ← ReciprocalEstimateSP(VSR[XB].word[i]) \\
    result.word[i] ← RoundToSP(RN, v) \\
    if(vxsnan_flag) then SetFX(VXSNAN) \\
    if(ox_flag) then SetFX(OX) \\
    if(ux_flag) then SetFX(UX) \\
    if(zx_flag) then SetFX(ZX) \\
    ex_flag ← ex_flag | (VE & vxsnan_flag) \\
    ex_flag ← ex_flag | (OE & ox_flag) \\
    ex_flag ← ex_flag | (UE & ux_flag) \\
    ex_flag ← ex_flag | (ZE & zx_flag) \\
end

if(ex_flag=0) then VSR[XT] ← result

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

For each vector element 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,

\[
\left| \frac{\text{estimate} - \frac{1}{\text{src}}}{\frac{1}{\text{src}}} \right| \leq \frac{1}{16384}
\]

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>-Infinity¹</td>
<td>ZX</td>
</tr>
<tr>
<td>+Zero</td>
<td>+Infinity¹</td>
<td>ZX</td>
</tr>
<tr>
<td>+Infinity</td>
<td>+Zero</td>
<td>None</td>
</tr>
<tr>
<td>SNaN</td>
<td>QNaN²</td>
<td>VXSNAN</td>
</tr>
<tr>
<td>QNaN</td>
<td>QNaN²</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 xvresp
\[
\begin{array}{cccccc}
0 & 32 & 64 & 96 & 127 \\
src = VSR[XB] \\
tgt = VSR[XT] \\
\end{array}
\]
### VSX Vector Round to Single-Precision Integer using round to Nearest Away XX2-form

<table>
<thead>
<tr>
<th>xvrspi</th>
<th>xt,xb</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-6</td>
<td>t</td>
</tr>
<tr>
<td>6-11</td>
<td>t1</td>
</tr>
<tr>
<td>12-15</td>
<td>t2</td>
</tr>
<tr>
<td>16-19</td>
<td>t3</td>
</tr>
<tr>
<td>20-23</td>
<td>t4</td>
</tr>
<tr>
<td>24-27</td>
<td>t5</td>
</tr>
<tr>
<td>28-31</td>
<td>t6</td>
</tr>
</tbody>
</table>

- Let `xt` be the value $32 \times t + t$.
- Let `xb` be the value $32 \times b + b$.

For each vector element $i$ from 0 to 3, do the following.

1. 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 to Nearest Away.
   - 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 xvrspi

<table>
<thead>
<tr>
<th>src = VSR[XB]</th>
</tr>
</thead>
<tbody>
<tr>
<td>SP</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>tgt = VSR[XT]</th>
</tr>
</thead>
<tbody>
<tr>
<td>SP</td>
</tr>
</tbody>
</table>

### VSX Vector Round to Single-Precision Integer Exact using Current rounding mode XX2-form

<table>
<thead>
<tr>
<th>xvrspic</th>
<th>xt,xb</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-6</td>
<td>t</td>
</tr>
<tr>
<td>6-11</td>
<td>t1</td>
</tr>
<tr>
<td>12-15</td>
<td>t2</td>
</tr>
<tr>
<td>16-19</td>
<td>t3</td>
</tr>
<tr>
<td>20-23</td>
<td>t4</td>
</tr>
<tr>
<td>24-27</td>
<td>t5</td>
</tr>
<tr>
<td>28-31</td>
<td>t6</td>
</tr>
</tbody>
</table>

- Let `xt` be the value $32 \times t + t$.
- Let `xb` be the value $32 \times b + b$.

For each vector element $i$ from 0 to 3, do the following.

1. 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>src = VSR[XB]</th>
</tr>
</thead>
<tbody>
<tr>
<td>SP</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>tgt = VSR[XT]</th>
</tr>
</thead>
<tbody>
<tr>
<td>SP</td>
</tr>
</tbody>
</table>
VSX Vector Round to Single-Precision Integer using round toward -Infinity XX2-form

\textbf{xvrspim} \quad XT, XB

\begin{array}{cccccccc}
60 & 6 & \text{I} & 16 & 21 & 185 & \text{bit} 13 & 31 \\
\end{array}

| XT | ← TX || T |
| XB | ← BX || B |
| ex_flag | ← 0b0 |

\text{do i=0 to 127 by 32}
\begin{align*}
\text{reset_xflags|} \\
\text{result\{i:i+31\} = RoundToSPIntegerFloor(VSR[XB]\{i:i+31\})} \\
\text{if(vxsnan_flag) then SetFX(VXSNAN)} \\
\text{ex_flag | ← ex_flag | (VE & vxsnan_flag)}
\end{align*}
\text{end}

\text{if( ex_flag = 0 ) then VSR[XT] ← result}

\text{Let XT be the value 32xTX + T.}
\text{Let XB be the value 32xBX + B.}

For each vector element \text{i} from 0 to 3, do the following.
\text{Let src be the single-precision floating-point operand in word element \text{i} of VSR[XB].}
\text{src is rounded to an integer using the rounding mode Round toward -Infinity.}
\text{The result is placed into word element \text{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].

\textbf{Special Registers Altered}
\textbf{FX VXSNAN}

VSX Vector Data Layout for xvrspim
\begin{array}{cccc}
\text{src = VSR[XB]} & \text{SP SP SP SP} \\
\text{tgt = VSR[XT]} & \text{SP SP SP SP} \\
0 & 32 & 64 & 96 & 127 \\
\end{array}

VSX Vector Round to Single-Precision Integer using round toward +Infinity XX2-form

\textbf{xvrspip} \quad XT, XB

\begin{array}{cccccccc}
60 & 6 & \text{I} & 16 & 21 & 189 & \text{bit} 13 & 31 \\
\end{array}

| XT | ← TX || T |
| XB | ← BX || B |
| ex_flag | ← 0b0 |

\text{do i=0 to 127 by 32}
\begin{align*}
\text{reset_xflags|} \\
\text{result\{i:i+31\} = RoundToSPIntegerCeil(VSR[XB]\{i:i+31\})} \\
\text{if(vxsnan_flag) then SetFX(VXSNAN)} \\
\text{ex_flag | ← ex_flag | (VE & vxsnan_flag)}
\end{align*}
\text{end}

\text{if( ex_flag = 0 ) then VSR[XT] ← result}

\text{Let XT be the value 32xTX + T.}
\text{Let XB be the value 32xBX + B.}

For each vector element \text{i} from 0 to 3, do the following.
\text{Let src be the single-precision floating-point operand in word element \text{i} of VSR[XB].}
\text{src is rounded to an integer using the rounding mode Round toward +Infinity.}
\text{The result is placed into word element \text{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].

\textbf{Special Registers Altered}
\textbf{FX VXSNAN}

VSX Vector Data Layout for xvrspip
\begin{array}{cccc}
\text{src = VSR[XB]} & \text{SP SP SP SP} \\
\text{tgt = VSR[XT]} & \text{SP SP SP SP} \\
0 & 32 & 64 & 96 & 127 \\
\end{array}
VSX Vector Round to Single-Precision Integer using round toward Zero XX2-form

\[ \text{xvrspliz} \quad \text{XT, XB} \]

\[
\begin{array}{ccccccc}
& 60 & T & \| & B & 153 & \text{BTX} \\
\hline
\text{XT} & \leftarrow & \text{TX} & \| & \text{T} \\
\text{XB} & \leftarrow & \text{BX} & \| & \text{B} \\
\text{ex_flag} & \leftarrow & 0b0 \\
\end{array}
\]

\begin{align*}
do \ i & \leftarrow 0 \text{ to } 127 \text{ by } 32 \\
\text{reset_xflags}() & \\
\text{result} & = \text{RoundToSFIntegerTrunc(VSR[XB]{i:i+31})} \\
(\text{if vxsnan_flag}) & \text{ then SetFX(VXSNAN)} \\
\text{ex_flag} & \leftarrow \text{ex_flag} \text{ | (VE & vxsnan_flag)} \\
\end{align*}

if( ex_flag = 0 ) then VSR[XT] \leftarrow \text{result}

Let XT be the value 32xTX + T.
Let XB be the value 32xBX + B.

For each vector element \( i \) 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

VSX Vector Reciprocal Square Root Estimate Double-Precision XX2-form

\[ \text{xvrsqrtedp} \quad \text{XT, XB} \]

\[
\begin{array}{ccccccc}
& 60 & T & \| & B & 202 & \text{B1X} \\
\hline
\text{XT} & \leftarrow & \text{TX} & \| & \text{T} \\
\text{XB} & \leftarrow & \text{BX} & \| & \text{B} \\
\text{ex_flag} & \leftarrow & 0b0 \\
\end{array}
\]

\begin{align*}
do \ i & \leftarrow 0 \text{ to } 127 \text{ by } 64 \\
\text{reset_xflags}() & \\
\text{v} & \{0:inf\} & \leftarrow & \text{RecipSquareEstimateDP(VSR[XB]{i:i+63})} \\
\text{result} & \{0:inf\} & \leftarrow & \text{RoundToDP}(\text{RN,v}) \\
(\text{if vxsqrt_flag}) & \text{ then SetFX(VXSQRT)} \\
(\text{if zx_flag}) & \text{ then SetFX(ZX)} \\
\text{ex_flag} & \leftarrow \text{ex_flag} \text{ | (VE & vxsqrt_flag)} \\
\text{ex_flag} & \leftarrow \text{ex_flag} \text{ | (ZE & zx_flag)} \\
\end{align*}

if( ex_flag = 0 ) then VSR[XT] \leftarrow \text{result}

Let XT be the value 32xTX + T.
Let XB be the value 32xBX + B.

For each vector element \( i \) 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,

\[
\left| \frac{\text{estimate} - \frac{1}{\sqrt{\text{src}}}}{\frac{1}{\sqrt{\text{src}}}} \right| \leq \frac{1}{16384}
\]

Operation with various special values of the operand is summarized below.

VSX Data Layout for xvrspliz
\[ src = \text{VSR[XB]} \]

\[
\begin{array}{cccc}
\text{SP} & \text{SP} & \text{SP} & \text{SP} \\
\hline
\text{tgt} & = & \text{VSR[XT]} \\
\end{array}
\]

\[
\begin{array}{cccc}
0 & 32 & 64 & 96 \\
\hline
\text{SP} & \text{SP} & \text{SP} & \text{SP} \\
\end{array}
\]

VSX Data Layout for xvrsqrtedp
\[ src = \text{VSR[XB]} \]

\[
\begin{array}{cccc}
\text{SP} & \text{SP} & \text{SP} & \text{SP} \\
\hline
\text{tgt} & = & \text{VSR[XT]} \\
\end{array}
\]

\[
\begin{array}{cccc}
0 & 32 & 64 & 96 \\
\hline
\text{SP} & \text{SP} & \text{SP} & \text{SP} \\
\end{array}
\]
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

<table>
<thead>
<tr>
<th>FX</th>
<th>ZX</th>
<th>VXSQRT</th>
<th>VXSNAN</th>
</tr>
</thead>
</table>

### VSR Data Layout for xvrsqrtedp

**src** = `VSR[XB]`

<table>
<thead>
<tr>
<th>0</th>
<th>64</th>
<th>127</th>
</tr>
</thead>
<tbody>
<tr>
<td>DP</td>
<td>DP</td>
<td></td>
</tr>
</tbody>
</table>

**tgt** = `VSR[XT]`

<table>
<thead>
<tr>
<th>0</th>
<th>64</th>
<th>127</th>
</tr>
</thead>
<tbody>
<tr>
<td>DP</td>
<td>DP</td>
<td></td>
</tr>
</tbody>
</table>
VSX Vector Reciprocal Square Root Estimate
Single-Precision XX2-form

`xvrsqtesp XT,XB`

### VSX Vector Reciprocal Square Root Estimate

| 0 | T | // | B | 138 | Bx
|---|---|----|---|----|---
| 60 | TX || T |
| 16 | BX || B |
| ex_flag | ← 0b0 |

do i=0 to 32 by 32
reset_flag();
v{0:inf} ← RecipSquareRootEstimateSP(VSR[XB]{i:i+31})
result{i:i+31} ← RoundToDP(RN,v)
  if(vxsnan_flag) then SetFX(VXSNAN)
  if(vxsqrt_flag) then SetFX(VXSQRT)
  if(zx_flag) then SetFX(ZX)
ex_flag ← ex_flag | (VE & vxsnan_flag)
ex_flag ← ex_flag | (VE & vxsqrt_flag)
ex_flag ← ex_flag | (ZE & zx_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

For each vector element 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,

\[
\left| \frac{\text{estimate} - \frac{1}{\sqrt{\text{src}}}}{\frac{1}{\sqrt{\text{src}}}} \right| \leq \frac{1}{16384}
\]

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>QNaN¹</td>
<td>VXSQRT</td>
</tr>
<tr>
<td>+Infinity</td>
<td>+Zero</td>
<td>None</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>SNaN</td>
<td>QNaN¹</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 Data Layout for xvrsqtesp**

```
src = VSR[XB]
tgt = VSR[XT]
```

<table>
<thead>
<tr>
<th>SP</th>
<th>SP</th>
<th>SP</th>
<th>SP</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>32</td>
<td>64</td>
<td>96</td>
</tr>
</tbody>
</table>

Operation with various special values of the operand is summarized below.
VSX Vector Square Root Double-Precision
XX2-form

xvsqrtdp  XT, XB

<table>
<thead>
<tr>
<th>60</th>
<th>6</th>
<th>//</th>
<th>16</th>
<th>21</th>
<th>203</th>
<th>B11</th>
</tr>
</thead>
<tbody>
<tr>
<td>XT</td>
<td>←</td>
<td>TX</td>
<td></td>
<td>T</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XB</td>
<td>←</td>
<td>BX</td>
<td></td>
<td>B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ex_flag</td>
<td>←</td>
<td>000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

do i = 0 to 127 by 64
reset_xflags();

v[0:inf] ← SquareRootDP(VSR[XB](i:i+63))
result[i:i+63] ← RoundToDP(RN, v)
if(vxsnan_flag) then SetFX(VXSNAN)
if(vxsqrt_flag) then SetFX(VXSQRT)
if(xx_flag) then SetFX(XX)

ex_flag ← ex_flag | (VE & vxsnan_flag)
ex_flag ← ex_flag | (VE & vxsqrt_flag)
ex_flag ← ex_flag | (XE & xx_flag)

end

if(ex_flag) then VSR[XT] ← result

Let XT be the value $32 \times TX + T$.
Let XB be the value $32 \times BX + B$.

For each vector element $i$ 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 127.

The intermediate result is rounded to double-precision using the rounding mode specified by RN.

See Table 50, “Scalar Floating-Point Intermediate Result Handling,” on page 515.

The result is placed into doubleword element $i$ of VSR[XT] in double-precision format.

See Table 98, “Vector Floating-Point Final Result,” on page 661.

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>
<thead>
<tr>
<th>VSR Data Layout for xvsqrtdp</th>
</tr>
</thead>
<tbody>
<tr>
<td>src = VSR[XB]</td>
</tr>
<tr>
<td>tgt = VSR[XT]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>64</th>
<th>127</th>
</tr>
</thead>
<tbody>
<tr>
<td>DP</td>
<td>DP</td>
</tr>
</tbody>
</table>


table 127.Action for xvsqrtdp

<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>exq_flag ← 1</td>
<td>exq_flag ← 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>exqnan_flag ← 1</td>
</tr>
</tbody>
</table>

Explanation:
src The double-precision floating-point value in doubleword element $i$ of VSR[XB] (where $i \in \{0, 1\}$).
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 127.Actions for xvsqrtdp
**VSX Vector Square Root Single-Precision XX2-form**

**xvsqrtsp** XT,XB

<table>
<thead>
<tr>
<th>0</th>
<th>60</th>
<th>6</th>
<th>11</th>
<th>15</th>
<th>19</th>
<th>21</th>
<th>139</th>
<th>202</th>
</tr>
</thead>
<tbody>
<tr>
<td>XT</td>
<td>← T</td>
<td></td>
<td>T</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XB</td>
<td>← B</td>
<td></td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ex_flag</td>
<td>← 0b0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Do i=0 to 127 by 32**

reset_vflags()

v[i:i+31] ← SquareRootSP(VSR[XB][i:i+31])

result[i:i+31] ← RoundToSP(RN,v)

if(vxsnan_flag) then SetFX(VXSNAN)

if(vxsqrt_flag) then SetFX(VXSQRT)

if(xx_flag) then SetFX(XX)

ex_flag ← ex_flag | (VE & vxsnan_flag)

ex_flag ← ex_flag | (VE & vxsqrt_flag)

ex_flag ← ex_flag | (XE & xx_flag)

end

if(ex_flag) then VSR[XT] ← result

Let XT be the value 32×TX + T.

Let XB be the value 32×BX + B.

For each vector element 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 128.

The intermediate result is rounded to single-precision using the rounding mode specified by RN.

<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>v←xvsqrt_flag←1</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.

See Table 50, “Scalar Floating-Point Intermediate Result Handling,” on page 515.

The result is placed into word element i of VSR[XT] in single-precision format.

See Table 98, “Vector Floating-Point Final Result,” on page 661.

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>XX</th>
<th>VXSNAN</th>
<th>VXSQRT</th>
</tr>
</thead>
</table>

**VSR Data Layout for xvsqrtsp**

<table>
<thead>
<tr>
<th>SP</th>
<th>SP</th>
<th>SP</th>
<th>SP</th>
</tr>
</thead>
<tbody>
<tr>
<td>tgt = VSR[XT]</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SP</th>
<th>32</th>
<th>64</th>
<th>96</th>
<th>127</th>
</tr>
</thead>
</table>

See Table 128, “Actions for xvsqrtsp”

752 Power ISA™ I
VSX Vector Subtract Double-Precision
XX3-form

\[
\text{xvsubdp XT,XA,XB}
\]

<table>
<thead>
<tr>
<th>i</th>
<th>0</th>
<th>60</th>
<th>T</th>
<th>A</th>
<th>B</th>
<th>104</th>
</tr>
</thead>
<tbody>
<tr>
<td>XT</td>
<td>← TX</td>
<td></td>
<td>T</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XA</td>
<td>← AX</td>
<td></td>
<td>A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XB</td>
<td>← BX</td>
<td></td>
<td>B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ex_flag</td>
<td>← 0b0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

do i=0 to 127 by 64
reset_xflags()
src1 ← VSR[XA]{i:i+63}
src2 ← VSR[XB]{i:i+63}
v0:v31f} ← AddDP(src1, NegateDP(src2)) |
result{i:i+63} ← RoundToDP(RN,v) |
if(vxsnan_flag) then SetFX(VXSNAN) |
if(vxisi_flag)  then SetFX(VXISI) |
if(ox_flag)     then SetFX(OX) |
if(ux_flag)     then SetFX(UX) |
if(xx_flag)     then SetFX(XX) |
ex_flag ← ex_flag | (VE & vxsnan_flag) |
ex_flag ← ex_flag | (VE & vxisi_flag) |
ex_flag ← ex_flag | (OE & ox_flag) |
ex_flag ← ex_flag | (UE & ux_flag) |
ex_flag ← ex_flag | (XE & xx_flag) |
end

if( ex_flag ) then VSR[XT] ← 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 vector element 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].
src2 is negated and added\(^1\) to src1, producing a sum having unbounded range and precision.
The sum is normalized\(^2\).

See Table 129.
The intermediate result is rounded to double-precision using the rounding mode specified by RN.

See Table 50, “Scalar Floating-Point Intermediate Result Handling,” on page 515.

---

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 129. Actions for xvsubdp

<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 ← dQNaN</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 ← 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>-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 ← 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 ← dQNaN</td>
<td>v ← src2</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 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**

The `vsubsp` instruction performs a single-precision floating-point subtraction operation. Here's a breakdown of the operation:

- **Input:**
  - **XT**: The value of `32xTX + T`
  - **XA**: The value of `32xAX + A`
  - **XB**: The value of `32xBX + B`

- **Loop:**
  - For each vector element `i` from 0 to 127 by 32

- **Operation:**
  - **src1**: `VSR[XA]{i:i+31}`
  - **src2**: `VSR[XB]{i:i+31}`
  - **v0**: `AddSP(src1, NegateSP(src2))`
  - **result**: `RoundToSP(RN, v)`
  - **Flag Handling:**
    - Set flags if `vxsnan_flag`, `vxisi_flag`, `ox_flag`, `ux_flag`, and `xx_flag`

- **Output:**
  - The result is placed into word element `i` of `VSR[XT]` in single-precision format.

- **Special Registers Altered:**
  - **FX, OX, UX, XX, VXSNAN, VXISI**

#### VSR Data Layout for `vsubsp`

```
src1 = VSR[XA]
src2 = VSR[XB]
tgt = VSR[XT]
```

#### Table 100, “Vector Floating-Point Final Result,” on page 661.

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**

---

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.
### 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>v ← src2</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>
<td>v ← 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>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).
- **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 Test for software Divide Double-Precision XX3-form

\[
xvtdivdp \quad BF,XA,XB
\]

<table>
<thead>
<tr>
<th>60</th>
<th>BF</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>125</th>
<th>CR[BF]</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>CR[BF]</td>
</tr>
</tbody>
</table>

\[
XA \leftarrow AX || A
XB \leftarrow BX || B
eq_flag \leftarrow 0b0
gt_flag \leftarrow 0b0
do i=0 to 127 by 64
\]

\[
src1 \leftarrow VSR[XA][i:i+63]
src2 \leftarrow VSR[XB][i:i+63]
e_a \leftarrow src1{1:11} - 1023
e_b \leftarrow src2{1:11} - 1023
fe_flag \leftarrow fe_flag | IsNaN(src1) | IsInf(src1) | IsNaN(src2) | IsInf(src2) | IsZero(src2) | (e_b <= -1022) | (e_b >= 1021) | (!IsZero(src1) & (e_a - e_b) >= 1023) | (!IsZero(src1) & (e_a - e_b) <= -1021) | (e_a <= -970)
fg_flag \leftarrow fg_flag | IsInf(src1) | IsInf(src2) | IsZero(src2) | IsDen(src2)
end
\]

fl_flag \leftarrow xvredp_error() <= 2^{-14}
CR[BF] \leftarrow 0b1 || fg_flag || fe_flag || 0b0

Let \( XA \) be the value \( 32 \times AX + A \). Let \( XB \) be the value \( 32 \times BX + B \).

\text{fe_flag} \text{ is initialized to 0.} 
\text{fg_flag} \text{ is initialized to 0.}

For each vector element \( i \) from 0 to 127, 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 \( src2 \).

\text{fe_flag} \text{ 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 \( -970 \).

\text{fg_flag} \text{ 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**

\text{CR[BF]}

**VSX Data Layout for xvtdivdp**

\[
src1 = VSR[XA]
\]

\[
.dword[0] \quad .dword[1]
\]

\[
src2 = VSR[XB]
\]

\[
.dword[0] \quad .dword[1]
\]
VSX Vector Test for software Divide
Single-Precision X3-form

xvtdivsp  BF,XA,XB

<table>
<thead>
<tr>
<th>60</th>
<th>BF</th>
<th>//</th>
<th>A</th>
<th>B</th>
<th>93</th>
<th>VREDP</th>
<th>001</th>
</tr>
</thead>
<tbody>
<tr>
<td>XA</td>
<td>← AX</td>
<td></td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XB</td>
<td>← BX</td>
<td></td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>eq_flag</td>
<td>← 0b0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>gt_flag</td>
<td>← 0b0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
\text{do } i = 0 \text{ to } 127 \text{ by 32} \\
\text{src1} & \leftarrow \text{VSR}[XA][i:i+31] \\
\text{src2} & \leftarrow \text{VSR}[XB][i:i+31] \\
\text{e}_a & \leftarrow \text{src1}[1:8] - 127 \\
\text{e}_b & \leftarrow \text{src2}[1:8] - 127 \\
\text{fe_flag} & \leftarrow \text{fe_flag} \mid \text{IsNaN(src1)} \mid \text{IsNaN(src2)} \mid \text{IsInf(src1)} \mid \text{IsInf(src2)} \mid \text{IsZero(src1)} \mid (\text{e}_b \leq -126) \mid (\text{e}_b \geq 125) \mid (\text{IsZero(src1)} \mid \text{IsInf(src1)} \mid (\text{e}_a - \text{e}_b) \geq 127) \mid (\text{IsZero(src1)} \mid \text{IsInf(src1)} \mid (\text{e}_a - \text{e}_b) \leq -125) \mid (\text{e}_a \leq -103) \\
\text{fg_flag} & \leftarrow \text{fg_flag} \mid \text{IsInf(src1)} \mid \text{IsInf(src2)} \mid \text{IsDen(src2)} \\
\text{fl_flag} & \leftarrow \text{VXREDP}() \leq 2^{-14} \\
\text{CR}[BF] & \leftarrow 0b1 \mid\mid \text{fg_flag} \mid\mid \text{fe_flag} \mid 0b0 \\
\]

Let XA be the value 32XAX + A.
Let XB be the value 32XBX + B.

fe_flag is initialized to 0.
fg_flag is initialized to 0.

For each vector element \(i\) 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(BF)

VSR Data Layout for xvtdivsp
	src1 = VSR[XA]

\[
\begin{array}{cccc}
\text{.word[0]} & \text{.word[1]} & \text{.word[2]} & \text{.word[3]} \\
0 & 6 & 64 & 96 & 127 \\
\end{array}
\]

src2 = VSR[XB]

\[
\begin{array}{cccc}
\text{.word[0]} & \text{.word[1]} & \text{.word[2]} & \text{.word[3]} \\
0 & 6 & 64 & 96 & 127 \\
\end{array}
\]
VSX Vector Test for software Square Root
Double-Precision XX2-form

xvtsqrtdp BF,XB

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>60</td>
<td>BF</td>
<td></td>
<td></td>
<td></td>
<td>B</td>
</tr>
<tr>
<td>6</td>
<td>9</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td></td>
</tr>
</tbody>
</table>

XB ← BX || B
fe_flag ← 0b0
fg_flag ← 0b0
do i=0 to 127 by 64
   src ← VSR[XB]{i:i+63}
   e_b ← src2[1:11] - 1023
   fe_flag ← fe_flag | IsNaN(src) | IsInf(src) | IsZero(src) | IsNeg(src) | ( e_a <= -970 )
   fg_flag ← fg_flag | IsInf(src) | IsZero(src) | IsDen(src)
end
fl_flag = xvrsqrtedp_error() <= 2-14
CR[BF] = 0b1 || fg_flag || fe_flag || 0b0

Let XB be the value 32×BX + B.

fe_flag is initialized to 0.
fg_flag is initialized to 0.

For each vector element 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[BF]

VSR Data Layout for xvtsqrtdp
src = VSR[XB]

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>64</td>
</tr>
<tr>
<td>127</td>
<td></td>
</tr>
</tbody>
</table>

VSX Vector Test for software Square Root
Single-Precision XX2-form

xvtsqrtsp BF,XB

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>60</td>
<td>BF</td>
<td></td>
<td></td>
<td></td>
<td>B</td>
</tr>
<tr>
<td>6</td>
<td>9</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td></td>
</tr>
</tbody>
</table>

XB ← BX || B
fe_flag ← 0b0
fg_flag ← 0b0
do i=0 to 127 by 32
   src ← VSR[XB]{i:i+31}
   e_b ← src2[1:8] - 127
   fe_flag ← fe_flag | IsNaN(src) | IsInf(src) | IsZero(src) | IsNeg(src) | ( e_a <= -103 )
   fg_flag ← fg_flag | IsInf(src) | IsZero(src) | IsDen(src)
end
fl_flag = xvrsqrtesp_error() <= 2-14
CR[BF] = 0b1 || fg_flag || fe_flag || 0b0

Let XB be the value 32×BX + B.

fe_flag is initialized to 0.
fg_flag is initialized to 0.

For each vector element 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[BF]

VSR Data Layout for xvtsqrtsp
src = VSR[XB]

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>32</td>
<td>64</td>
<td>96</td>
</tr>
<tr>
<td>127</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Let $XB$ be the sum $32 \times BX + B$.
Let $XT$ be the sum $32 \times TX + T$.
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 $0x0000_0000_0000_0000$.

### DCMX bit Data Class

<table>
<thead>
<tr>
<th>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

#### VSR Data Layout for xvtstcdcp

<table>
<thead>
<tr>
<th>src</th>
<th>VSR[32×BX+B].dword[0]</th>
<th>VSR[32×BX+B].dword[1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>tgt</td>
<td>VSR[32×TX+T].dword[0]</td>
<td>VSR[32×TX+T].dword[1]</td>
</tr>
</tbody>
</table>
VSX Vector Test Data Class Single-Precision

**XX2-form**

```
xvtstdcsp XT,XB,DCMX
```

Let $X_B$ be the sum $32 \times X_B + B$.

Let $X_T$ be the sum $32 \times X_T + T$.

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 $VSR[X_B]$.

If $src$ matches one of the 7 possible data classes specified by $DCMX$ (Data Class Mask), the contents of word element $i$ of $VSR[X_T]$ are set to $0xFFFF_FFFF$. Otherwise, the contents of word element $i$ of $VSR[X_T]$ 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

```
```

```
```

if MSR.VSX=0 then VSX_Unavailable();
VSX Vector Extract Exponent
Double-Precision XX2-form

\[ xvxexpdp \ XT, XB \]

<table>
<thead>
<tr>
<th>src VSR[XB].dword[0]</th>
<th>VSR[XB].dword[1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>tgt VSR[XT].dword[0]</td>
<td>VSR[XT].dword[1]</td>
</tr>
</tbody>
</table>

\[
\text{if MSR.VSX} = 0 \text{ then VSX_Unavailable()}
\]

\[
do \ i = 0 \ to \ 1
\]

\[
\text{do } i = 0 \text{ to } 1
\]
\[
\begin{align*}
\text{src} & \leftarrow \text{VSR}[32 \times B + B].\text{dword}[i] \\
\text{VSR}[32 \times T + T].\text{dword}[i] & \leftarrow \text{EXTZ64(src.bit[1:11])}
\end{align*}
\]

\[
\text{end}
\]

\[
\text{let } XT \text{ be the sum } 32 \times T + T.
\]
\[
\text{let } XB \text{ be the sum } 32 \times B + B.
\]

For each integer value \(i\) from 0 to 1, do the following.

Let \(\text{src}\) be the double-precision floating-point value in doubleword element \(i\) of \(\text{VSR}[XB]\).

The value of the exponent field in \(\text{src}\) is placed into doubleword element \(i\) of \(\text{VSR}[XT]\) in unsigned integer format.

Special Registers Altered:
None

VSR Data Layout for xvxexpdp

\[
\begin{array}{cccc}
0 & 60 & T & 11 \\
6 & 0 & B & 21 \\
\end{array}
\]

\[
\begin{array}{cccc}
0 & 60 & T & 11 \\
6 & 0 & B & 21 \\
\end{array}
\]

VSX Vector Extract Exponent Single-Precision
XX2-form

\[ xvxexpsp \ XT, XB \]

\[
\begin{array}{cccc}
0 & 60 & T & 8 \\
6 & 0 & B & 21 \\
\end{array}
\]

\[
\begin{array}{cccc}
0 & 60 & T & 8 \\
6 & 0 & B & 21 \\
\end{array}
\]

\[
\text{if MSR.VSX} = 0 \text{ then VSX_Unavailable()}
\]

\[
do \ i = 0 \text{ to } 3
\]

\[
\text{do } i = 0 \text{ to } 3
\]
\[
\begin{align*}
\text{src} & \leftarrow \text{VSR}[32 \times B + B].\text{word}[i] \\
\text{VSR}[32 \times T + T].\text{word}[i] & \leftarrow \text{EXTZ32(src.bit[1:8])}
\end{align*}
\]

\[
\text{end}
\]

Let \(XT\) be the sum \(32 \times T + T\).
Let \(XB\) be the sum \(32 \times B + B\).

For each integer value \(i\) from 0 to 3, do the following.

Let \(\text{src}\) be the single-precision floating-point value in word element \(i\) of \(\text{VSR}[XB]\).

The value of the exponent field in \(\text{src}\) is placed into word element \(i\) of \(\text{VSR}[XT]\) in unsigned integer format.

Special Registers Altered:
None

VSR Data Layout for xvxexpsp

\[
\begin{array}{cccc}
0 & 3 & 6 & 11 \\
0 & 3 & 26 & 49 \\
0 & 6 & 11 & 16 \\
6 & 11 & 16 & 21 \\
9 & 6 & 11 & 21 \\
\end{array}
\]

\[
\begin{array}{cccc}
0 & 3 & 6 & 11 \\
0 & 3 & 26 & 49 \\
0 & 6 & 11 & 16 \\
6 & 11 & 16 & 21 \\
9 & 6 & 11 & 21 \\
\end{array}
\]
Chapter 7. Vector-Scalar Floating-Point Operations

VSX Vector Extract Significand
Double-Precision XX2-form

xvxsigdp XT,XB

<table>
<thead>
<tr>
<th>60</th>
<th>T</th>
<th>1</th>
<th>B</th>
<th>475</th>
<th>B[X]</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()

do i = 0 to 1
    src ← VSR[32×BX+B].dword[i]
    exponent ← EXTZ(src.bit[1:11])
    fraction ← EXTZ64(src.bit[12:63])
    if (exponent ≠ 0) & (exponent ≠ 2047) then
        fraction ← fraction | 0x0010_0000_0000_0000
    VSR[32×TX+T].dword[i] ← fraction
end

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>
<tr>
<td>0</td>
<td>64</td>
<td>127</td>
</tr>
</tbody>
</table>

VSX Vector Extract Significand
Single-Precision XX2-form

xvxsigsp XT,XB

<table>
<thead>
<tr>
<th>60</th>
<th>T</th>
<th>9</th>
<th>B</th>
<th>475</th>
<th>B[X]</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()

do i = 0 to 3
    src ← VSR[32×BX+B].word[i]
    exponent ← EXTZ(src.bit[1:8])
    fraction ← EXTZ32(src.bit[9:31])
    if (exponent ≠ 0) & (exponent ≠ 255) then
        fraction ← fraction | 0x0080_0000
    VSR[32×TX+T].word[i] ← fraction
end

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

VSR Data Layout for xvxsigsp

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>32</td>
<td>96</td>
<td>127</td>
<td></td>
</tr>
</tbody>
</table>
VSX Vector Byte-Reverse Doubleword XX2-form

\[ \text{xxbrd\ XT, XB} \]

<table>
<thead>
<tr>
<th>60</th>
<th>T</th>
<th>23</th>
<th>B</th>
<th>475</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()

\[
\begin{align*}
\text{do } i &= 0 \text{ to } 1 \\
\text{do } j &= 0 \text{ to } 7 \\
VSR[32\times TX + T].dword[i].byte[j] &\leftarrow VSR[32\times BX + B].dword[i].byte[7-j] \\
\end{align*}
\]

Special Registers Altered:

None

VSX Vector Byte-Reverse Halfword XX2-form

\[ \text{xxbrh\ XT, XB} \]

<table>
<thead>
<tr>
<th>60</th>
<th>T</th>
<th>7</th>
<th>B</th>
<th>475</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()

\[
\begin{align*}
\text{do } i &= 0 \text{ to } 7 \\
VSR[32\times TX + T].hword[i].byte[0] &\leftarrow VSR[32\times BX + B].hword[i].byte[1] \\
VSR[32\times TX + T].hword[i].byte[1] &\leftarrow VSR[32\times BX + B].hword[i].byte[0] \\
\end{align*}
\]

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 7, do the following.

The contents of byte 0 of halfword element \(i\) of VSR[XB] are placed into byte 1 of halfword element \(i\) of VSR[XT].

The contents of byte 1 of halfword element \(i\) of VSR[XB] are placed into byte 0 of halfword element \(i\) of VSR[XT].

Special Registers Altered:

None
**VSX Vector Byte-Reverse Quadword XX2-form**

\[
\text{xxbrq} \quad \text{XT},\text{XB}
\]

<table>
<thead>
<tr>
<th>60</th>
<th>T</th>
<th>31</th>
<th>B</th>
<th>475</th>
<th>0</th>
</tr>
</thead>
</table>

Let \( \text{XT} \) be the value \( 32 \times \text{T} + \text{T} \).
Let \( \text{XB} \) be the value \( 32 \times \text{B} + \text{B} \).

For each integer value \( i \) from 0 to 15, do the following.

- The contents of byte sub-element \( 15-i \) of \( \text{VSR}[\text{XB}] \) are placed into byte sub-element \( i \) of \( \text{VSR}[\text{XT}] \).

**Special Registers Altered:**
None

---

**VSX Vector Byte-Reverse Word XX2-form**

\[
\text{xxbrw} \quad \text{XT},\text{XB}
\]

<table>
<thead>
<tr>
<th>60</th>
<th>T</th>
<th>15</th>
<th>B</th>
<th>475</th>
<th>0</th>
</tr>
</thead>
</table>

if MSR.VSX=0 then VSX_Unavailable();
doi = 0 to 3

do j = 0 to 3

\[
\text{VSR}[32 \times \text{T} + \text{T}].\text{word}[i].\text{byte}[j] \leftarrow \text{VSR}[32 \times \text{B} + \text{B}].\text{word}[3-j].\text{byte}[j]
\]
endo

Let \( \text{XT} \) be the value \( 32 \times \text{T} + \text{T} \).
Let \( \text{XB} \) be the value \( 32 \times \text{B} + \text{B} \).

For each integer value \( i \) from 0 to 3, do the following.

- The contents of byte 3 of word element \( i \) of \( \text{VSR}[\text{XB}] \) are placed into byte 0 of word element \( i \) of \( \text{VSR}[\text{XT}] \).
- The contents of byte 2 of word element \( i \) of \( \text{VSR}[\text{XB}] \) are placed into byte 1 of word element \( i \) of \( \text{VSR}[\text{XT}] \).
- The contents of byte 1 of word element \( i \) of \( \text{VSR}[\text{XB}] \) are placed into byte 2 of word element \( i \) of \( \text{VSR}[\text{XT}] \).
- The contents of byte 0 of word element \( i \) of \( \text{VSR}[\text{XB}] \) are placed into byte 3 of word element \( i \) of \( \text{VSR}[\text{XT}] \).

**Special Registers Altered:**
None
VSX Vector Extract Unsigned Word XX2-form

**xxextractuw** XT,XB,UIM

<table>
<thead>
<tr>
<th>60</th>
<th>T</th>
<th>UIM</th>
<th>B</th>
<th>165</th>
<th>BX</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>12</td>
<td>16</td>
<td>21</td>
<td>10</td>
</tr>
</tbody>
</table>

Let XT be the value \(32 \times T + T\).
Let XB be the value \(32 \times B + B\).

The contents of byte elements UI M: UI M+3 of VSR[XB] are placed into word element 1 of VSR[XT]. The contents of the remaining word elements of VSR[XT] are set to 0.

If the value of UI M is greater than 12, the results are undefined.

**Special Registers Altered:**

None

VSX Vector Insert Word XX2-form

**xxinsertw** XT,XB,UIM

<table>
<thead>
<tr>
<th>60</th>
<th>T</th>
<th>UIM</th>
<th>B</th>
<th>181</th>
<th>BX</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>12</td>
<td>16</td>
<td>21</td>
<td>11</td>
</tr>
</tbody>
</table>

Let XT be the value \(32 \times T + T\).
Let XB be the value \(32 \times B + B\).

The contents of word element 1 of VSR[XB] are placed into byte elements UI M: UI M+3 of VSR[XT]. The contents of the remaining byte elements of VSR[XT] are not modified.

If the value of UI M is greater than 12, the results are undefined.

**Special Registers Altered:**

None
**VSX Logical AND XX3-form**

- **xxland**
  - XT,XA,XB

<table>
<thead>
<tr>
<th>60</th>
<th>T</th>
<th>A</th>
<th>B</th>
<th>130</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SB</td>
</tr>
</tbody>
</table>


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**

- `src1 = VSR[XA]`
- `src2 = VSR[XB]`
- `tgt = VSR[XT]`

---

**VSX Logical AND with Complement XX3-form**

- **xxlandc**
  - XT,XA,XB

<table>
<thead>
<tr>
<th>60</th>
<th>T</th>
<th>A</th>
<th>B</th>
<th>138</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SB</td>
</tr>
</tbody>
</table>


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

**VSR Data Layout for xxlandc**

- `src1 = VSR[XA]`
- `src2 = VSR[XB]`
- `tgt = VSR[XT]`
**VSX Logical Equivalence XX3-form**

**xxleqv** \( XT,XA,XB \)

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
</table>
| 0 | 60 | T | A | B | 186 | VSR[32×TX+T] 268216

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**

- \( src = VSR[XA] \)
- \( src = VSR[XB] \)
- \( tgt = VSR[XT] \)

---

**VSX Logical NAND XX3-form**

**xxlnand** \( XT,XA,XB \)

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
</table>
| 0 | 60 | T | A | B | 178 | VSR[32×TX+T] 268216

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

**VSR Data Layout for xxlnand**

- \( src = VSR[XA] \)
- \( src = VSR[XB] \)
- \( tgt = VSR[XT] \)
**VSX Logical OR with Complement XX3-form**

\[
\text{xxlorc} \quad XT,XA,XB
\]

<table>
<thead>
<tr>
<th>60</th>
<th>6</th>
<th>T</th>
<th>A</th>
<th>B</th>
<th>170</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td></td>
<td>120</td>
</tr>
</tbody>
</table>

\[\text{VSR}[32\times X + T] \leftarrow \text{VSR}[32\times A + A] \mid \neg \text{VSR}[32\times B + B] \]

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 \( \text{VSR}[XA] \) are ORed with the complement of the contents of \( \text{VSR}[XB] \) and the result is placed into \( \text{VSR}[XT] \).

**Special Registers Altered:**
None

\textbf{VSR Data Layout for xxlorc}

- \( \text{src1} = \text{VSR}[XA] \)
- \( \text{src2} = \text{VSR}[XB] \)
- \( \text{tgt} = \text{VSR}[XT] \)

---

**VSX Logical NOR XX3-form**

\[
\text{xxlnor} \quad XT,XA,XB
\]

<table>
<thead>
<tr>
<th>60</th>
<th>6</th>
<th>T</th>
<th>A</th>
<th>B</th>
<th>162</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td></td>
<td>120</td>
</tr>
</tbody>
</table>

\[\text{VSR}[32\times X + T] \leftarrow \neg \left( \text{VSR}[32\times A + A] \mid \text{VSR}[32\times B + B] \right) \]

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 \( \text{VSR}[XA] \) are ORed with the contents of \( \text{VSR}[XB] \) and the complemented result is placed into \( \text{VSR}[XT] \).

**Special Registers Altered:**
None

\textbf{VSR Data Layout for xxlnor}

- \( \text{src1} = \text{VSR}[XA] \)
- \( \text{src2} = \text{VSR}[XB] \)
- \( \text{tgt} = \text{VSR}[XT] \)
**VSX Logical OR XX3-form**

xxlor  
XT,XA,XB

<table>
<thead>
<tr>
<th>60</th>
<th>6</th>
<th>T</th>
<th>A</th>
<th>B</th>
<th>146</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
\text{VSR}[32 \times T + T] \leftarrow \text{VSR}[32 \times A + A] \mid \text{VSR}[32 \times B + B]
\]

Let \( T \) be the value \( 32 \times T + T \).
Let \( A \) be the value \( 32 \times A + A \).
Let \( B \) be the value \( 32 \times B + B \).

The contents of \( \text{VSR}[A] \) are ORed with the contents of \( \text{VSR}[B] \) and the result is placed into \( \text{VSR}[T] \).

**Special Registers Altered**

None

**VSR Data Layout for xxlor**

src1 = VSR[XA]

tgt = VSR[XT]

<table>
<thead>
<tr>
<th>60</th>
<th>6</th>
<th>T</th>
<th>A</th>
<th>B</th>
<th>146</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**VSX Logical XOR XX3-form**

xxlxor  
XT,XA,XB

<table>
<thead>
<tr>
<th>60</th>
<th>6</th>
<th>T</th>
<th>A</th>
<th>B</th>
<th>154</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
\text{VSR}[32 \times T + T] \leftarrow \text{VSR}[32 \times A + A] \oplus \text{VSR}[32 \times B + B]
\]

Let \( T \) be the value \( 32 \times T + T \).
Let \( A \) be the value \( 32 \times A + A \).
Let \( B \) be the value \( 32 \times B + B \).

The contents of \( \text{VSR}[A] \) are exclusive-ORed with the contents of \( \text{VSR}[B] \) and the result is placed into \( \text{VSR}[T] \).

**Special Registers Altered**

None

**VSR Data Layout for xxlxor**

src1 = VSR[XA]

tgt = VSR[XT]

<table>
<thead>
<tr>
<th>60</th>
<th>6</th>
<th>T</th>
<th>A</th>
<th>B</th>
<th>154</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
VSX Merge High Word 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 $32 \times BX + B$.

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

VSX Data Layout for xxmrglw

```
src1 = VSR[XA]
    .word[0] .word[1] unused unused
src2 = VSR[XB]
    .word[0] .word[1] unused unused
tgt = VSR[XT]
```

if MSR.VSX=0 then VSX_Unavailable()}

VSR[32\times TX+T].word[0] \leftarrow VSR[32\times AX+A].word[0]
VSR[32\times TX+T].word[1] \leftarrow VSR[32\times BX+B].word[0]
VSR[32\times TX+T].word[2] \leftarrow VSR[32\times AX+A].word[1]
VSR[32\times TX+T].word[3] \leftarrow VSR[32\times BX+B].word[1]

VSX Merge Low Word 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 $32 \times BX + B$.

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

VSX Data Layout for xxmrglw

```
src1 = VSR[XA]
src2 = VSR[XB]
tgt = VSR[XT]
```

if MSR.VSX=0 then VSX_Unavailable()}

VSR[32\times TX+T].word[0] \leftarrow VSR[32\times AX+A].word[2]
VSR[32\times TX+T].word[1] \leftarrow VSR[32\times BX+B].word[2]
VSR[32\times TX+T].word[2] \leftarrow VSR[32\times AX+A].word[3]
VSR[32\times TX+T].word[3] \leftarrow VSR[32\times BX+B].word[3]
VSX Vector Permute XX3-form

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]

\[
\text{do } i = 0 \text{ to } 15 \\
\text{idx} ← \text{pcv.byte}[i].bit[3:7] \\
\text{VSR}[32\times TX + T].byte[i] ← \text{src.byte}[\text{idx}]
\]

Let XA be the value \(32\times AX + A\).
Let XB be the value \(32\times BX + B\).
Let XT be the value \(32\times TX + T\).

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

VSX Vector Permute Right-indexed XX3-form

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]

\[
\text{do } i = 0 \text{ to } 15 \\
\text{idx} ← \text{pcv.byte}[i].bit[3:7] \\
\text{VSR}[32\times TX + T].byte[i] ← \text{src.byte}[31-\text{idx}]
\]

Let XA be the value \(32\times AX + A\).
Let XB be the value \(32\times BX + B\).
Let XT be the value \(32\times TX + T\).

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
**VSX Permute Doubleword Immediate XX3-form**

\[
\text{xxpermdi } \quad \text{XT,XA,XB,DM}
\]

<table>
<thead>
<tr>
<th>60</th>
<th>T</th>
<th>A</th>
<th>B</th>
<th>0</th>
<th>DM</th>
<th>10</th>
<th>VSR[TX+T]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>54</td>
<td>12</td>
<td>80</td>
</tr>
</tbody>
</table>

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×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 Mnemonic Equivalent To**

- **xxspltd** T,A,0  xxpermdi T,A,A,0b00
- **xxspltd** T,A,1  xxpermdi T,A,A,0b11
- **xxmrghd** T,A,B  xxpermdi T,A,B,0b00
- **xxmrld** T,A,B  xxpermdi T,A,B,0b11
- **xxswapd** T,A  xxpermdi T,A,A,0b10

**VSR Data Layout for xxpermdi**

\[
\begin{align*}
\text{src1 = VSR[XA]} & \quad .dword[0] \quad .dword[1] \\
\text{src2 = VSR[XB]} & \quad .dword[0] \quad .dword[1] \\
\text{tgt = VSR[XT]} & \quad .dword[0] \quad .dword[1]
\end{align*}
\]

**VSX Select XX4-form**

\[
\text{xxsel } \quad \text{XT,XA,XB XC}
\]

<table>
<thead>
<tr>
<th>60</th>
<th>T</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>28</th>
<th>VSR[32×TX+T]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>26</td>
<td>80</td>
</tr>
</tbody>
</table>

if MSR.VSX=0 then VSX_Unavailable()

\[
\begin{align*}
\text{VSR[32×TX+T].dword[0]} & \leftarrow \text{VSR[32×AX+A].dword[DM.bit[0]]} \\
\text{VSR[32×TX+T].dword[1]} & \leftarrow \text{VSR[32×BX+B].dword[DM.bit[1]]}
\end{align*}
\]

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 bit of VSR[XC] that contains the value 0, the corresponding bit of VSR[XA] is placed into the corresponding bit of VSR[XT]. Otherwise, the corresponding bit of VSR[XB] is placed into the corresponding bit of VSR[XT].

**Special Registers Altered**

None

**VSR Data Layout for xxsel**

\[
\begin{align*}
\text{src1 = VSR[XA]} & \quad .dword[0] \\
\text{src2 = VSR[XB]} & \quad .dword[0] \\
\text{src3 = VSR[XC]} & \quad .dword[0] \\
\text{tgt = VSR[XT]} & \quad .dword[0]
\end{align*}
\]

0  127
VSX Shift Left Double by Word Immediate
XX3-form

xxsldwi XT,XA,XB,SHW

| 0 | 60 | T | A | B | 0 | SHW | 2 | WBRX | 2 | 2 |

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 the source vector be the concatenation of the contents of VSR[XT] followed by the contents of VSR[XB]. Words SHW:SHW+3 of the source vector are placed into VSR[XT].

Special Registers Altered
None

VSR Data Layout for xxsldwi

src1 = VSR[XA]


src2 = VSR[XB]


tgt = VSR[XT]


VSX Splat Word XX2-form

xxspltw XT,XB,UIM

| 0 | 60 | T | IM | UIM | B | 164 | X |

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 $32 \times TX + T$.
Let XB be the value $32 \times BX + B$.

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

src = VSR[XB]


tgt = VSR[XT]


VSX Vector Splat Immediate Byte X-form

xxspltib XT,IMM8

| 0 | 60 | T | 0 | IMM8 | 360 | X |

if TX=0 & MSR.VSX=0 then VSX_Unavailable()
if TX=1 & MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 15
   VSR[32×TX+T].byte[i] ← IMM8
end

Let XT be the sum $32 \times TX + T$.

The value IMM8 is copied into each byte element of VSR[XT].

Special Registers Altered:
None
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.

\[
\text{If } (\text{FRB})_{1:11} < 897 \text{ and } (\text{FRB})_{1:63} > 0 \text{ then}
\]
\[
\text{Do}
\]
\[
\text{If } \text{FPSCR}_{UE} = 0 \text{ then goto Disabled Exponent Underflow}
\]
\[
\text{If } \text{FPSCR}_{UE} = 1 \text{ then goto Enabled Exponent Underflow}
\]
\[
\text{End}
\]
\[
\text{If } (\text{FRB})_{1:11} > 1150 \text{ and } (\text{FRB})_{1:11} < 2047 \text{ then}
\]
\[
\text{Do}
\]
\[
\text{If } \text{FPSCR}_{OE} = 0 \text{ then goto Disabled Exponent Overflow}
\]
\[
\text{If } \text{FPSCR}_{OE} = 1 \text{ then goto Enabled Exponent Overflow}
\]
\[
\text{End}
\]
\[
\text{If } (\text{FRB})_{1:11} > 896 \text{ and } (\text{FRB})_{1:11} < 1151 \text{ then goto Normal Operand}
\]
\[
\text{If } (\text{FRB})_{1:63} = 0 \text{ then goto Zero Operand}
\]
\[
\text{If } (\text{FRB})_{1:11} = 2047 \text{ then}
\]
\[
\text{Do}
\]
\[
\text{If } (\text{FRB})_{12:63} = 0 \text{ then goto Infinity Operand}
\]
\[
\text{If } (\text{FRB})_{12} = 1 \text{ then goto QNaN Operand}
\]
\[
\text{If } (\text{FRB})_{12} = 0 \text{ and } (\text{FRB})_{13:63} > 0 \text{ then goto SNaN Operand}
\]
\[
\text{End}
\]

Disabled Exponent Underflow:

\[
\text{sign } \leftarrow (\text{FRB})_{0}
\]
\[
\text{If } (\text{FRB})_{1:11} = 0 \text{ then}
\]
\[
\text{Do}
\]
\[
\text{exp } \leftarrow -1022
\]
\[
\text{frac}_{0:52} \leftarrow 0b0 || (\text{FRB})_{12:63}
\]
\[
\text{End}
\]
\[
\text{If } (\text{FRB})_{1:11} > 0 \text{ then}
\]
\[
\text{Do}
\]
\[
\text{exp } \leftarrow (\text{FRB})_{1:11} - 1023
\]
\[
\text{frac}_{0:52} \leftarrow 0b1 || (\text{FRB})_{12:63}
\]
\[
\text{End}
\]

Denormalize operand:

\[
\text{G } || \text{R } || \text{X } \leftarrow 0b000
\]
\[
\text{Do while } \text{exp } < -126
\]
\[
\text{exp } \leftarrow \text{exp } + 1
\]
\[
\text{frac}_{0:52} || \text{G } || \text{R } || \text{X } \leftarrow 0b0 || \text{frac}_{0:52} || \text{G } || (\text{R } || \text{X})
\]
\[
\text{End}
\]
\[
\text{FPSCR}_{UX} \leftarrow (\text{frac}_{24:52} || \text{G } || \text{R } || \text{X}) > 0
\]
\[
\text{Round Single(sign,exp,frac}_{0:52} \text{,G,R,X)}
\]
\[
\text{FPSCR}_{XX} \leftarrow \text{FPSCR}_{UX} || \text{FPSCR}_{FI}
\]
\[
\text{If } \text{frac}_{0:52} = 0 \text{ then}
\]
\[
\text{Do}
\]
\[
\text{FRT}_{0} \leftarrow \text{sign}
\]
\[
\text{FRT}_{1:63} \leftarrow 0
\]
If sign = 0 then FPSCRFPREF ^= "+ zero"
If sign = 1 then FPSCRFPREF ^= "- zero"
End
If frac0:52 > 0 then
Do
  If frac0 = 1 then
    Do
      If sign = 0 then FPSCRFPREF ^= "+ normal number"
      If sign = 1 then FPSCRFPREF ^= "- normal number"
    End
  If frac0 = 0 then
    Do
      If sign = 0 then FPSCRFPREF ^= "+ denormalized number"
      If sign = 1 then FPSCRFPREF ^= "- 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
Done

Enabled Exponent Underflow:
FPSCRUX ^= 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)
FPSCRXX ^= FPSCRXX | FPSCR_FI
exp ^= exp + 192
FRT0 ^= sign
FRT1:11 ^= exp + 1023
FRT12:63 ^= frac1:52
If sign = 0 then FPSCRFPREF ^= "+ normal number"
If sign = 1 then FPSCRFPREF ^= "- 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 ^= 0xFFF0_0000_0000_0000
    If (FRB)0 = 0 then FPSCRFPREF ^= "+ infinity"
    If (FRB)0 = 1 then FPSCRFPREF ^= "- infinity"
  End
If FPSCR_RN = 0b01 then /* Round toward Zero */
  Do

"
If (FRB)₀ = 0 then FRT ← 0x47EF_FFFF_E000_0000
If (FRB)₀ = 1 then FRT ← 0xC7EF_FFFF_E000_0000
If (FRB)₀ = 0 then FPSCRₚ₟RF ← “+” normal number
If (FRB)₀ = 1 then FPSCRₚ₟RF ← “−” normal number

End
If FPSCRₚ₟RN = 0b10 then /* Round toward +Infinity */
Do
If (FRB)₀ = 0 then FRT ← 0x7FF0_0000_0000_0000
If (FRB)₀ = 1 then FRT ← 0xC7EF_FFFF_E000_0000
If (FRB)₀ = 0 then FPSCRₚ₟RF ← “+” infinity
If (FRB)₀ = 1 then FPSCRₚ₟RF ← “−” normal number
End
If FPSCRₚ₟RN = 0b11 then /* Round toward -Infinity */
Do
If (FRB)₀ = 0 then FRT ← 0x47EF_FFFF_E000_0000
If (FRB)₀ = 1 then FRT ← 0xFFF0_0000_0000_0000
If (FRB)₀ = 0 then FPSCRₚ₟RF ← “+” normal number
If (FRB)₀ = 1 then FPSCRₚ₟RF ← “−” infinity
End
FPSCRₚ₟FR ← undefined
FPSCRₚ₟FI ← 1
FPSCRₚ₟XX ← 1
Done

Enabled Exponent Overflow:

sign ← (FRB)₀
exp ← (FRB)₁:₁₁ - 1023
frac₀:₅₂ ← 0b1 || (FRB)₁₂:₆₃
Round Single(sign,exp,frac₀:₅₂,0,0,0)
FPSCRₚ₟XX ← FPSCRₚ₟XX | FPSCRₚ₟FI

Enabled Overflow:

FPSCRₚ₟OX ← 1
exp ← exp - 192
FRT₀ ← sign
FRT₁:₁₁ ← exp + 1023
FRT₁₂:₆₃ ← frac₁:₅₂
If sign = 0 then FPSCRₚ₟RF ← “+” normal number
If sign = 1 then FPSCRₚ₟RF ← “−” normal number
Done

Zero Operand:

FRT ← (FRB)
If (FRB)₀ = 0 then FPSCRₚ₟RF ← “+” zero
If (FRB)₀ = 1 then FPSCRₚ₟RF ← “−” zero
FPSCRₚ₟FRF₁ ← 0b00
Done

Infinity Operand:

FRT ← (FRB)
If (FRB)₀ = 0 then FPSCRₚ₟RF ← “+” infinity
If (FRB)₀ = 1 then FPSCRₚ₟RF ← “−” infinity
FPSCRₚ₟FRF₁ ← 0b00
Done

QNaN Operand:

FRT ← (FRB)₀:₃₄ || "₉" 90
FPSCRₚ₟RF ← “QNaN”
FPSCRₚ₟FRF₁ ← 0b00
Done
SNaN Operand:
\[
\text{FPSCR}_{\text{X} \times \text{SNaN}} \leftarrow 1
\]
If FPSCR\textit{VE} = 0 then
\[
\text{Do}
\]
\[
\text{FRT}_{0:11} \leftarrow (\text{FRB})_{0:11}
\]
\[
\text{FRT}_{12} \leftarrow 1
\]
\[
\text{FRT}_{13:63} \leftarrow (\text{FRB})_{13:34} || 290
\]
\[
\text{FPSCR}_{\text{FPRF}} \leftarrow "QNaN"
\]
\[
\text{End}
\]
\[
\text{FPSCR}_{\text{FR FI}} \leftarrow 0b00
\]
Done

Normal Operand:
\[
\text{sign} \leftarrow (\text{FRB})_0
\]
\[
\text{exp} \leftarrow (\text{FRB})_{1:11} - 1023
\]
\[
\text{frac}_{0:52} \leftarrow 0b1 || (\text{FRB})_{12:63}
\]
Round Single(sign,exp,frac\textit{0:52},0,0,0)
\[
\text{FPSCR}_{\text{XX}} \leftarrow \text{FPSCR}_{\text{XX}} \mid \text{FPSCR}_{\text{FI}}
\]
If exp > 127 and FPSCR\textit{OE} = 0 then go to Disabled Exponent Overflow
If exp > 127 and FPSCR\textit{OE} = 1 then go to Enabled Overflow
\[
\text{FRT}_0 \leftarrow \text{sign}
\]
\[
\text{FRT}_{1:11} \leftarrow \text{exp} + 1023
\]
\[
\text{FRT}_{12:63} \leftarrow \text{frac}_{1:52}
\]
If sign = 0 then FPSCR\textit{FPRF} \leftarrow "+ normal number"
If sign = 1 then FPSCR\textit{FPRF} \leftarrow "- normal number"
Done

Round Single(sign,exp,frac\textit{0:52},G,R,X):
\[
\text{inc} \leftarrow 0
\]
\[
\text{lsb} \leftarrow \text{frac}_{23}
\]
\[
\text{gbit} \leftarrow \text{frac}_{24}
\]
\[
\text{rbit} \leftarrow \text{frac}_{25}
\]
\[
\text{xbit} \leftarrow (\text{frac}_{26:52}||G||R||X) > 0
\]
If FPSCR\textit{RN} = 0b00 then /* Round to Nearest */
\[
\text{Do}
\]
\[
/* \text{comparisons ignore u bits} */
\]
\[
\text{If sign} \parallel \text{lsb} \parallel \text{gbit} \parallel \text{rbit} \parallel \text{xbit} = 0bu11uu then inc \leftarrow 1
\]
\[
\text{If sign} \parallel \text{lsb} \parallel \text{gbit} \parallel \text{rbit} \parallel \text{xbit} = 0bu011u then inc \leftarrow 1
\]
\[
\text{If sign} \parallel \text{lsb} \parallel \text{gbit} \parallel \text{rbit} \parallel \text{xbit} = 0bu01u1 then inc \leftarrow 1
\]
\[
\text{End}
\]
If FPSCR\textit{RN} = 0b10 then /* Round toward + Infinity */
\[
\text{Do}
\]
\[
/* \text{comparisons ignore u bits} */
\]
\[
\text{If sign} \parallel \text{lsb} \parallel \text{gbit} \parallel \text{rbit} \parallel \text{xbit} = 0b01uu then inc \leftarrow 1
\]
\[
\text{If sign} \parallel \text{lsb} \parallel \text{gbit} \parallel \text{rbit} \parallel \text{xbit} = 0b0uu1u then inc \leftarrow 1
\]
\[
\text{If sign} \parallel \text{lsb} \parallel \text{gbit} \parallel \text{rbit} \parallel \text{xbit} = 0b0uuu1 then inc \leftarrow 1
\]
\[
\text{End}
\]
If FPSCR\textit{RN} = 0b11 then /* Round toward - Infinity */
\[
\text{Do}
\]
\[
/* \text{comparisons ignore u bits} */
\]
\[
\text{If sign} \parallel \text{lsb} \parallel \text{gbit} \parallel \text{rbit} \parallel \text{xbit} = 0b1uu1u then inc \leftarrow 1
\]
\[
\text{If sign} \parallel \text{lsb} \parallel \text{gbit} \parallel \text{rbit} \parallel \text{xbit} = 0b1uuu1 then inc \leftarrow 1
\]
\[
\text{End}
\]
frac\textit{0:23} \leftarrow frac\textit{0:23} + inc
If carry\_out = 1 then
\[
\text{Do}
\]
\[
\text{frac}_{0:23} \leftarrow 0b1 || \text{frac}_{0:22}
\]
\[
\text{exp} \leftarrow \text{exp} + 1
\]
\[
\text{End}
\]
frac\textit{24:52} \leftarrow 290
\[
\text{FPSCR}_{\text{FR}} \leftarrow \text{inc}
\]
\[
\text{FPSCR}_{\text{FI}} \leftarrow \text{gbit} \mid \text{rbit} \mid \text{xbit}
\]
Return
A.2 Floating-Point Convert to Integer Model

The following describes algorithmically the operation of the Floating Convert To Integer instructions.

if Floating Convert To Integer Word then do
    round_mode ← FPSCR<31>
    tgt_precision ← “32-bit signed integer”
end

if Floating Convert To Integer Word Unsigned then do
    round_mode ← FPSCR<31>
    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<31>
    tgt_precision ← “64-bit signed integer”
end

if Floating Convert To Integer Doubleword Unsigned then do
    round_mode ← FPSCR<31>
    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 frac<0:64> ← 0b01 || (FRB)<12:63> || 110 /* normal */
if (FRB)<1:11> 0 then frac<0:64> ← 0b00 || (FRB)<12:63> || 110 /* denormal */
gbit || rbit || xbit ← 0b000
do i=1,63-exp /* do the loop 0 times if exp = 63 */
    frac<0:64> || gbit || rbit || xbit ← 0b0 || frac<0:64> || gbit || (rbit | xbit)
end

Round Integer( sign, frac<0:64>, gbit, rbit, xbit, round_mode )
if sign = 1 then frac<0:64> ← ¬frac<0:64> + 1 /* needed leading 0 for -2^64<(FRB)<2^63 */
if tgt_precision = “32-bit signed integer” and frac0:64 > 2^{31}-1 then
goto Large Operand
if tgt_precision = “64-bit signed integer” and frac0:64 > 2^{63}-1 then
goto Large Operand
if tgt_precision = “32-bit signed integer” and frac0:64 < -2^{31} then
goto Large Operand
if tgt_precision = “64-bit signed integer” and frac0:64 < -2^{63} then
goto Large Operand
if tgt_precision = “32-bit unsigned integer” & frac0:64 > 2^{32}-1 then
goto Large Operand
if tgt_precision = “64-bit unsigned integer” & frac0:64 > 2^{64}-1 then
goto Large Operand
if tgt_precision = “32-bit unsigned integer” & frac0:64 < 0 then
goto Large Operand
if tgt_precision = “64-bit unsigned integer” & frac0:64 < 0 then
goto Large Operand

FPSCRXX ← FPSCRXX | FPSCR_FI

if tgt_precision = “32-bit signed integer” then FRT ← 0xUUUU_UUUU || frac33:64
if tgt_precision = “32-bit unsigned integer” then FRT ← 0xUUUU_UUUU || frac33:64
if tgt_precision = “64-bit signed integer” then FRT ← frac1:64
if tgt_precision = “64-bit unsigned integer” then FRT ← frac1:64
FPSCR_PPFF ← ObUUUU

done

Round Integer( sign, frac0:64, gbit, rbit, xbit, round_mode ):

inc ← 0
if round_mode = 0b00 then do /* Round to Nearest */
if sign || frac64 || gbit || rbit || xbit = 0bU11UU then inc ← 1
if sign || frac64 || gbit || rbit || xbit = 0bU011U then inc ← 1
if sign || frac64 || gbit || rbit || xbit = 0bU01U1 then inc ← 1
end
if round_mode = 0b10 then do /* Round toward +Infinity */
if sign || frac64 || gbit || rbit || xbit = 0b0U11U then inc ← 1
if sign || frac64 || gbit || rbit || xbit = 0b00U1U then inc ← 1
if sign || frac64 || gbit || rbit || xbit = 0b00U1U then inc ← 1
end
if round_mode = 0b11 then do /* Round toward -Infinity */
if sign || frac64 || gbit || rbit || xbit = 0b1U11U then inc ← 1
if sign || frac64 || gbit || rbit || xbit = 0b10U1U then inc ← 1
if sign || frac64 || gbit || rbit || xbit = 0b10U1U then inc ← 1
end
frac0:64 ← frac0:64 + inc
FPSCR_FP ← inc
FPSCR_FI ← gbit | rbit | xbit
return

Infinity Operand:

FPSCR_FP ← 0b0
FPSCR_FI ← 0b0
FPSCR_VXX | FPSCR_VX ← 0b1
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_FPRF ← 0bUUUUUU
done

SNaN Operand:
FPSCR_FPR ← 0b0
FPSCR_FPI ← 0b0
FPSCR_VXSNAN ← 0b1
FPSCR_VXCVI ← 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
end
FPSCR_FPRF ← 0bUUUUUU
done

QNaN Operand:
FPSCR_FPR ← 0b0
FPSCR_FPI ← 0b0
FPSCR_VXCVI ← 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
end
FPSCR_FPRF ← 0bUUUUUU
done

Large Operand:
FPSCR_FPR ← 0b0
FPSCR_FPI ← 0b0
FPSCR_VXCVI ← 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_FPRF ← 0bUUUUUU
done
A.3 Floating-Point Convert from Integer Model

The following describes algorithmically the operation of the *Floating Convert From Integer* instructions.

```plaintext
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

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 ← frac52
gbit ← frac53
rbit ← frac54
xbit ← frac55:63 > 0
end

if round_mode = 0b00 then do                /* Round to Nearest */
  if sign || lsb || gbit || rbit || xbit = 0bU1UU1 then inc ← 1
  if sign || lsb || gbit || rbit || xbit = 0bU01U1 then inc ← 1
  if sign || lsb || gbit || rbit || xbit = 0bU011U 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
  frac0:23 ← frac0:23 + inc
else /* tgt_precision = "double-precision" */
  frac0:52 ← frac0:52 + inc

if carry_out = 1 then exp ← exp + 1

FPSCRFR ← inc
FPSCRFI ← gbit | rbit | xbit
FPSCRXX ← FPSCRXX | 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 /* exp < 0; |value| < 1*/
If \((FRB)_{1:11} > 1023\) then goto Small Operand /* exp < 0; |value| < 1*/
If \((FRB)_{1:11} > 1074\) then goto Large Operand /* exp > 51; integral value */

\[
\begin{align*}
sign & \leftarrow (FRB)_0 \\
\exp & \leftarrow (FRB)_{1:11} - 1023 \quad \text{/* exp - bias */}
frac_{0:52} & \leftarrow 0b1 \parallel (FRB)_{12:63} \\
gbit \parallel rbit \parallel xbit & \leftarrow 0b000
\end{align*}
\]

\[
\text{Do } i = 1, 52 - \exp \quad \frac{frac_{0:52}}{gbit \parallel rbit \parallel xbit} \leftarrow 0b0 \parallel \frac{frac_{0:52}}{gbit \parallel (rbit \mid xbit)}
\]

End

Round Integer \((\text{sign, frac}_{0:52}, \text{gbit, rbit, xbit})\):

\[
\text{Do } i = 2, 52 - \exp \quad \frac{frac_{0:52}}{frac_{1:52}} \leftarrow 0b0
\]

End

If \(frac_0 = 1\), then \(\exp \leftarrow \exp + 1\)
Else \(frac_{0:52} \leftarrow frac_{1:52} \parallel 0b0\)

\[
\begin{align*}
FRT_0 & \leftarrow \text{sign} \\
FRT_{1:11} & \leftarrow \exp + 1023 \\
FRT_{12:63} & \leftarrow frac_{1:52}
\end{align*}
\]

If \((FRT)_0 = 0\) then \(\text{FPSCR}_{\text{FR1}} \leftarrow "+ normal number"
Else \(\text{FPSCR}_{\text{FR1}} \leftarrow "- normal number"
\text{FPSCR}_{\text{FR1}} \leftarrow 0b00
Done

Round Integer\((\text{sign, frac}_{0:52}, \text{gbit, rbit, xbit})\):

\[
\begin{align*}
\text{inc} & \leftarrow 0 \\
\text{If inst = Floating Round to Integer Nearest then} & \quad /* \text{ties away from zero */}
\text{Do } /* \text{comparisons ignore u bits */}
\text{If sign || frac}_{52} || \text{gbit || rbit || xbit} = 0b0u1uu \text{then inc} \leftarrow 1
\text{End}
\text{If inst = Floating Round to Integer Plus then} & \quad /* \text{comparisons ignore u bits */}
\text{Do } /* \text{comparisons ignore u bits */}
\text{If sign || frac}_{52} || \text{gbit || rbit || xbit} = 0b0u1uu \text{then inc} \leftarrow 1
\text{If sign || frac}_{52} || \text{gbit || rbit || xbit} = 0b0uu1u \text{then inc} \leftarrow 1
\text{If sign || frac}_{52} || \text{gbit || rbit || xbit} = 0b0uuu1 \text{then inc} \leftarrow 1
\text{End}
\text{If inst = Floating Round to Integer Minus then} & \quad /* \text{comparisons ignore u bits */}
\text{Do } /* \text{comparisons ignore u bits */}
\text{If sign || frac}_{52} || \text{gbit || rbit || xbit} = 0b1u1uu \text{then inc} \leftarrow 1
\text{If sign || frac}_{52} || \text{gbit || rbit || xbit} = 0b1uu1u \text{then inc} \leftarrow 1
\text{If sign || frac}_{52} || \text{gbit || rbit || xbit} = 0b1uuu1 \text{then inc} \leftarrow 1
\text{End}
\frac{frac_{0:52}}{frac_{0:52} \leftarrow \frac{frac_{0:52} \leftarrow \text{inc}}{Return}}
\]

784 Power ISA™ I
Infinity Operand:
\[
\text{FRT} \leftarrow (\text{FRB}) \\
\text{If } (\text{FRB})_0 = 0 \text{ then } \text{FPSCR}_{\text{FPFRF}} \leftarrow + \text{ infinity} \\
\text{If } (\text{FRB})_0 = 1 \text{ then } \text{FPSCR}_{\text{FPFRF}} \leftarrow - \text{ infinity} \\
\text{FPSCR}_{\text{FR Fi}} \leftarrow 0b00 \\
\text{Done}
\]

SNaN Operand:
\[
\text{FPSCR}_{\text{VXSNAN}} \leftarrow 1 \\
\text{If FPSCR}_{VE} = 0 \text{ then} \\
\text{Do} \\
\text{FRT} \leftarrow (\text{FRB}) \\
\text{FRT}_{12} \leftarrow 1 \\
\text{FPSCR}_{\text{FPFRF}} \leftarrow \text{QNaN} \\
\text{End} \\
\text{FPSCR}_{\text{FR Fi}} \leftarrow 0b00 \\
\text{Done}
\]

QNaN Operand:
\[
\text{FRT} \leftarrow (\text{FRB}) \\
\text{FPSCR}_{\text{FPFRF}} \leftarrow \text{QNaN} \\
\text{FPSCR}_{\text{FR Fi}} \leftarrow 0b00 \\
\text{Done}
\]

Zero Operand:
\[
\text{If } (\text{FRB})_0 = 0 \text{ then} \\
\text{Do} \\
\text{FRT} \leftarrow 0x0000_0000_0000_0000 \\
\text{FPSCR}_{\text{FPFRF}} \leftarrow + \text{ zero} \\
\text{End} \\
\text{Else} \\
\text{Do} \\
\text{FRT} \leftarrow 0x8000_0000_0000_0000 \\
\text{FPSCR}_{\text{FPFRF}} \leftarrow - \text{ zero} \\
\text{End} \\
\text{FPSCR}_{\text{FR Fi}} \leftarrow 0b00 \\
\text{Done}
\]

Small Operand:
\[
\text{If inst = Floating Round to Integer Nearest and} \\
(\text{FRB})_{1:11} < 1022 \text{ then goto Zero Operand} \\
\text{If inst = Floating Round to Integer Toward Zero} \\
\text{then goto Zero Operand} \\
\text{If inst = Floating Round to Integer Plus and } (\text{FRB})_0 = 1 \text{ then goto Zero Operand} \\
\text{If inst = Floating Round to Integer Minus and} \\
(\text{FRB})_0 = 0 \text{ then goto Zero Operand} \\
\text{If } (\text{FRB})_0 = 0 \text{ then} \\
\text{Do} \\
\text{FRT} \leftarrow 0x3FF0_0000_0000_0000 \\
/\star \text{ value } = 1.0 \star/ \\
\text{FPSCR}_{\text{FPFRF}} \leftarrow + \text{ normal number} \\
\text{End} \\
\text{Else} \\
\text{Do} \\
\text{FRT} \leftarrow 0xBFF0_0000_0000_0000 \\
/\star \text{ value } = -1.0 \star/ \\
\text{FPSCR}_{\text{FPFRF}} \leftarrow - \text{ normal number} \\
\text{End} \\
\text{FPSCR}_{\text{FR Fi}} \leftarrow 0b00 \\
\text{Done}
\]

Large Operand:
\[
\text{FRT} \leftarrow (\text{FRB})
\]
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)(ijkl), 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.

\[
\begin{align*}
p &= (f & a & i & ¬e) | (j & a & ¬i) | (b & ¬a) \\
q &= (g & a & i & ¬e) | (k & a & ¬i) | (c & ¬a) \\
r &= d \\
s &= (j & ¬a & e & ¬i) | (f & ¬i & ¬e) | (f & ¬a & ¬e) | (e & i) \\
t &= (k & ¬a & e & ¬i) | (g & ¬i & ¬e) | (g & ¬a & ¬e) | (a & i) \\
u &= h \\
v &= a | e | i \\
w &= (¬a & j & ¬i) | (e & i) | a \\
x &= (¬a & k & ¬i) | (a & i) | e \\
y &= m
\end{align*}
\]

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>aei</th>
<th>pqr stu v wxy</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>bcd fgh 0 jkm</td>
</tr>
<tr>
<td>001</td>
<td>bcd fgh 1 00m</td>
</tr>
<tr>
<td>010</td>
<td>bcd jkh 0 01m</td>
</tr>
<tr>
<td>011</td>
<td>bcd jkh 1 11m</td>
</tr>
<tr>
<td>100</td>
<td>jkd fgh 1 10m</td>
</tr>
<tr>
<td>101</td>
<td>fgd 01h 1 11m</td>
</tr>
<tr>
<td>110</td>
<td>jkd 00h 1 11m</td>
</tr>
<tr>
<td>111</td>
<td>00d 11h 1 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 131 on page 789, 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.

\[
\begin{align*}
a &= (¬s & v & w) | (t & v & w & s) | (v & w & ¬x) \\
b &= (p & s & x & ¬t) | (p & ¬w) | (p & ¬v) \\
c &= (q & s & x & ¬t) | (q & ¬w) | (q & ¬v) \\
d &= x \\
e &= (v & ¬w & x) | (s & v & w & x) | (¬t & v & w & x) \\
f &= (p & t & v & w & x & ¬s) | (s & ¬x & v) | (s & ¬v) \\
g &= (q & t & w & v & x & ¬s) | (t & ¬x & v) | (t & ¬v) \\
h &= u \\
i &= (t & v & w & x) | (s & v & w & x) | (v & ¬w & ¬x) \\
j &= (p & ¬s & ¬t & w & v) | (s & v & ¬w & x) | (p & w & ¬x & v) | (w & ¬v) \\
k &= (q & ¬s & ¬t & v & w) | (t & v & ¬w & x) | (q & v & w & ¬x) | (x & ¬v) \\
m &= y
\end{align*}
\]

Alternatively, the following table can be used to perform the translation. A combination of five bits in the DPD encoding (leftmost column) is 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.

<table>
<thead>
<tr>
<th>ae</th>
<th>pqr stu v wxy</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>bcd fgh 0 jkm</td>
</tr>
<tr>
<td>01</td>
<td>bcd jkh 0 01m</td>
</tr>
<tr>
<td>10</td>
<td>bcd jkh 1 11m</td>
</tr>
<tr>
<td>11</td>
<td>jkd fgh 1 10m</td>
</tr>
<tr>
<td>100</td>
<td>jkd 00h 1 11m</td>
</tr>
<tr>
<td>101</td>
<td>fgd 01h 1 11m</td>
</tr>
<tr>
<td>110</td>
<td>00d 11h 1 11m</td>
</tr>
</tbody>
</table>
The full translation of the 10-bit DPD to a 3-digit BCD number is shown in Table 132 on page 790. 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>vwxst</th>
<th>abcd</th>
<th>efgh</th>
<th>ijkm</th>
</tr>
</thead>
<tbody>
<tr>
<td>0----</td>
<td>0pqr</td>
<td>0stu</td>
<td>0wxy</td>
</tr>
<tr>
<td>100--</td>
<td>0pqr</td>
<td>0stu</td>
<td>100y</td>
</tr>
<tr>
<td>101--</td>
<td>0pqr</td>
<td>100u</td>
<td>0sty</td>
</tr>
<tr>
<td>110--</td>
<td>100r</td>
<td>0stu</td>
<td>0pqy</td>
</tr>
<tr>
<td>11100</td>
<td>100r</td>
<td>100u</td>
<td>0pqy</td>
</tr>
<tr>
<td>11101</td>
<td>100r</td>
<td>0pqu</td>
<td>100y</td>
</tr>
<tr>
<td>11110</td>
<td>0pqr</td>
<td>100u</td>
<td>100y</td>
</tr>
<tr>
<td>11111</td>
<td>100r</td>
<td>100u</td>
<td>100y</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DPD Code</th>
<th>BCD Value</th>
<th>DPD Code</th>
<th>BCD Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x06E</td>
<td>888</td>
<td>0x0EE</td>
<td>988</td>
</tr>
<tr>
<td>(0x16E)</td>
<td></td>
<td>(0x1EE)</td>
<td></td>
</tr>
<tr>
<td>(0x26E)</td>
<td></td>
<td>(0x2EE)</td>
<td></td>
</tr>
<tr>
<td>(0x36E)</td>
<td></td>
<td>(0x3EE)</td>
<td></td>
</tr>
<tr>
<td>0x06F</td>
<td>889</td>
<td>0x0EF</td>
<td>989</td>
</tr>
<tr>
<td>(0x16F)</td>
<td></td>
<td>(0x1EF)</td>
<td></td>
</tr>
<tr>
<td>(0x26F)</td>
<td></td>
<td>(0x2EF)</td>
<td></td>
</tr>
<tr>
<td>(0x36F)</td>
<td></td>
<td>(0x3EF)</td>
<td></td>
</tr>
<tr>
<td>0x07E</td>
<td>898</td>
<td>0x0FE</td>
<td>998</td>
</tr>
<tr>
<td>(0x17E)</td>
<td></td>
<td>(0x1FE)</td>
<td></td>
</tr>
<tr>
<td>(0x27E)</td>
<td></td>
<td>(0x2FE)</td>
<td></td>
</tr>
<tr>
<td>(0x37E)</td>
<td></td>
<td>(0x3FE)</td>
<td></td>
</tr>
<tr>
<td>0x07F</td>
<td>899</td>
<td>0x0FF</td>
<td>999</td>
</tr>
<tr>
<td>(0x17F)</td>
<td></td>
<td>(0x1FF)</td>
<td></td>
</tr>
<tr>
<td>(0x27F)</td>
<td></td>
<td>(0x2FF)</td>
<td></td>
</tr>
<tr>
<td>(0x37F)</td>
<td></td>
<td>(0x3FF)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
</tr>
<tr>
<td>00</td>
<td>000</td>
<td>001</td>
<td>002</td>
</tr>
<tr>
<td>01</td>
<td>010</td>
<td>011</td>
<td>012</td>
</tr>
<tr>
<td>02</td>
<td>020</td>
<td>021</td>
<td>022</td>
</tr>
<tr>
<td>03</td>
<td>030</td>
<td>031</td>
<td>032</td>
</tr>
<tr>
<td>04</td>
<td>040</td>
<td>041</td>
<td>042</td>
</tr>
<tr>
<td>05</td>
<td>050</td>
<td>051</td>
<td>052</td>
</tr>
<tr>
<td>06</td>
<td>060</td>
<td>061</td>
<td>062</td>
</tr>
<tr>
<td>07</td>
<td>070</td>
<td>071</td>
<td>072</td>
</tr>
<tr>
<td>08</td>
<td>080</td>
<td>081</td>
<td>082</td>
</tr>
<tr>
<td>09</td>
<td>090</td>
<td>091</td>
<td>092</td>
</tr>
</tbody>
</table>

Table 131: BCD-to-DPD translation

- Appendix B. Densely Packed Decimal

789
Version 3.0 B

Table 132: DPD-to-BCD translation
00_
01_
02_
03_
04_
05_
06_
07_
08_
09_
0A_
0B_
0C_
0D_
0E_
0F_
10_
11_
12_
13_
14_
15_
16_
17_
18_
19_
1A_
1B_
1C_
1D_
1E_
1F_
20_
21_
22_
23_
24_
25_
26_
27_
28_
29_
2A_
2B_
2C_
2D_
2E_
2F_
30_
31_
32_
33_
34_
35_
36_
37_
38_
39_
3A_
3B_
3C_
3D_
3E_
3F_

790

0
000
010
020
030
040
050
060
070
100
110
120
130
140
150
160
170
200
210
220
230
240
250
260
270
300
310
320
330
340
350
360
370
400
410
420
430
440
450
460
470
500
510
520
530
540
550
560
570
600
610
620
630
640
650
660
670
700
710
720
730
740
750
760
770

1
001
011
021
031
041
051
061
071
101
111
121
131
141
151
161
171
201
211
221
231
241
251
261
271
301
311
321
331
341
351
361
371
401
411
421
431
441
451
461
471
501
511
521
531
541
551
561
571
601
611
621
631
641
651
661
671
701
711
721
731
741
751
761
771

2
002
012
022
032
042
052
062
072
102
112
122
132
142
152
162
172
202
212
222
232
242
252
262
272
302
312
322
332
342
352
362
372
402
412
422
432
442
452
462
472
502
512
522
532
542
552
562
572
602
612
622
632
642
652
662
672
702
712
722
732
742
752
762
772

Power ISA™ I

3
003
013
023
033
043
053
063
073
103
113
123
133
143
153
163
173
203
213
223
233
243
253
263
273
303
313
323
333
343
353
363
373
403
413
423
433
443
453
463
473
503
513
523
533
543
553
563
573
603
613
623
633
643
653
663
673
703
713
723
733
743
753
763
773

4
004
014
024
034
044
054
064
074
104
114
124
134
144
154
164
174
204
214
224
234
244
254
264
274
304
314
324
334
344
354
364
374
404
414
424
434
444
454
464
474
504
514
524
534
544
554
564
574
604
614
624
634
644
654
664
674
704
714
724
734
744
754
764
774

5
005
015
025
035
045
055
065
075
105
115
125
135
145
155
165
175
205
215
225
235
245
255
265
275
305
315
325
335
345
355
365
375
405
415
425
435
445
455
465
475
505
515
525
535
545
555
565
575
605
615
625
635
645
655
665
675
705
715
725
735
745
755
765
775

6
006
016
026
036
046
056
066
076
106
116
126
136
146
156
166
176
206
216
226
236
246
256
266
276
306
316
326
336
346
356
366
376
406
416
426
436
446
456
466
476
506
516
526
536
546
556
566
576
606
616
626
636
646
656
666
676
706
716
726
736
746
756
766
776

7
007
017
027
037
047
057
067
077
107
117
127
137
147
157
167
177
207
217
227
237
247
257
267
277
307
317
327
337
347
357
367
377
407
417
427
437
447
457
467
477
507
517
527
537
547
557
567
577
607
617
627
637
647
657
667
677
707
717
727
737
747
757
767
777

8
008
018
028
038
048
058
068
078
108
118
128
138
148
158
168
178
208
218
228
238
248
258
268
278
308
318
328
338
348
358
368
378
408
418
428
438
448
458
468
478
508
518
528
538
548
558
568
578
608
618
628
638
648
658
668
678
708
718
728
738
748
758
768
778

9
009
019
029
039
049
059
069
079
109
119
129
139
149
159
169
179
209
219
229
239
249
259
269
279
309
319
329
339
349
359
369
379
409
419
429
439
449
459
469
479
509
519
529
539
549
559
569
579
609
619
629
639
649
659
669
679
709
719
729
739
749
759
769
779

A
080
090
082
092
084
094
086
096
180
190
182
192
184
194
186
196
280
290
282
292
284
294
286
296
380
390
382
392
384
394
386
396
480
490
482
492
484
494
486
496
580
590
582
592
584
594
586
596
680
690
682
692
684
694
686
696
780
790
782
792
784
794
786
796

B
081
091
083
093
085
095
087
097
181
191
183
193
185
195
187
197
281
291
283
293
285
295
287
297
381
391
383
393
385
395
387
397
481
491
483
493
485
495
487
497
581
591
583
593
585
595
587
597
681
691
683
693
685
695
687
697
781
791
783
793
785
795
787
797

C
800
810
820
830
840
850
860
870
900
910
920
930
940
950
960
970
802
812
822
832
842
852
862
872
902
912
922
932
942
952
962
972
804
814
824
834
844
854
864
874
904
914
924
934
944
954
964
974
806
816
826
836
846
856
866
876
906
916
926
936
946
956
966
976

D
801
811
821
831
841
851
861
871
901
911
921
931
941
951
961
971
803
813
823
833
843
853
863
873
903
913
923
933
943
953
963
973
805
815
825
835
845
855
865
875
905
915
925
935
945
955
965
975
807
817
827
837
847
857
867
877
907
917
927
937
947
957
967
977

E
880
890
808
818
088
098
888
898
980
990
908
918
188
198
988
998
882
892
828
838
288
298
(888)
(898)
982
992
928
938
388
398
(988)
(998)
884
894
848
858
488
498
(888)
(898)
984
994
948
958
588
598
(988)
(998)
886
896
868
878
688
698
(888)
(898)
986
996
968
978
788
798
(988)
(998)

F
881
891
809
819
089
099
889
899
981
991
909
919
189
199
989
999
883
893
829
839
289
299
(889)
(899)
983
993
929
939
389
399
(989)
(999)
885
895
849
859
489
499
(889)
(899)
985
995
949
959
589
599
(989)
(999)
887
897
869
879
689
699
(889)
(899)
987
997
969
979
789
799
(989)
(999)


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[f], bcctr[l]) with the appropriate operands.

C.2.2 Simple Branch Mnemonics

Instructions using one of the mnemonics in Table 133 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>
<thead>
<tr>
<th>Branch Semantics</th>
<th>LR not Set</th>
<th>LR Set</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>b Relative</td>
<td>bca Absolute</td>
</tr>
<tr>
<td>Branch unconditionally</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Branch if CRBI=1</td>
<td>bt</td>
<td>bta</td>
</tr>
<tr>
<td>Branch if CRBI=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 CRBI=1</td>
<td>bdnzr</td>
<td>bdnzta</td>
</tr>
<tr>
<td>Decrement CTR, branch if CTR nonzero and CRBI=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 CRBI=1</td>
<td>bdzt</td>
<td>bdzta</td>
</tr>
<tr>
<td>Decrement CTR, branch if CTR zero and CRBI=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).
   \[
   \text{bdnz target} \quad (\text{equivalent to:} \quad \text{bc 16,0,target})
   \]

2. Same as (1) but branch only if CTR is nonzero and condition in CR0 is “equal”.
   \[
   \text{bdnzr eq,target} \quad (\text{equivalent to:} \quad \text{bc 8,2,target})
   \]

3. Same as (2), but “equal” condition is in CR5.
   \[
   \text{bdnz 4<cr5>eq,target} \quad (\text{equivalent to:} \quad \text{bc 8,22,target})
   \]
4. Branch if bit 59 of CR is 0.
   \[\text{bf 27,target} \quad \text{(equivalent to: bc 4,27,target)}\]

5. Same as (4), but set the Link Register. This is a form of conditional “call”.
   \[\text{bfl 27,target} \quad \text{(equivalent to: bcl 4,27,target)}\]

### C.2.3 Branch Mnemonics Incorporating Conditions

In the mnemonics defined in Table 134, the test of a bit in a Condition Register field is encoded in the mnemonic.

Instructions using the mnemonics in Table 134 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 \(\text{bclr}\) or \(\text{bcctr}\) and the BH operand is specified.

A standard set of codes has been adopted for the most common combinations of branch conditions.

#### Example
1. Branch if CR0 reflects condition “not equal”.
   \[\text{bne target} \quad \text{(equivalent to: bc 4,2,target)}\]

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".

\[ \text{bgta} \ cr4,\text{target} \quad \text{(equivalent to: bcla 12,17,\text{target})} \]

4. Same as (3), but target address is in the Count Register.

\[ \text{bgctr} \ cr4 \quad \text{(equivalent to: bcctr 12,17,0)} \]

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.

\[ \text{blt}+ \quad \text{target} \]

2. Same as (1), but target address is in the Link Register and the branch should be predicted not to be taken.

\[ \text{bltlr}- \]
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>cnot bx,by</td>
<td>crnor 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.
   crcl 4×cr3+so (equivalent to: crxor 15,15,15)

3. Same as (2), but SO bit to be cleared is in CR3.

4. Invert the EQ bit.
   cnot eq.eq (equivalent to: crnor 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.
   cnot 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.

subi Rx,Ry,value (equivalent to: addi Rx,Ry,−value)
subis Rx,Ry,value (equivalent to: addis Rx,Ry,−value)
subic Rx,Ry,value (equivalent to: addic Rx,Ry,−value)
subic Rx,Ry,value (equivalent to: addicRx,Ry,−value)

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.

sub Rx,Ry,Rz (equivalent to: subf Rx,Rz,Ry)
subc Rx,Ry,Rz (equivalent to: subfc Rx,Rz,Ry)
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>Table 136: Doubleword compare mnemonics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation</td>
</tr>
<tr>
<td>Compare doubleword immediate</td>
</tr>
<tr>
<td>Compare doubleword</td>
</tr>
<tr>
<td>Compare logical doubleword immediate</td>
</tr>
<tr>
<td>Compare logical doubleword</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: cmpli 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>Table 137: Word compare mnemonics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation</td>
</tr>
<tr>
<td>Compare word immediate</td>
</tr>
<tr>
<td>Compare word</td>
</tr>
<tr>
<td>Compare logical word immediate</td>
</tr>
<tr>
<td>Compare logical word</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: cmpi 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: cmpi 0,0,Rx,Ry)
C.6 Trap Mnemonics

The mnemonics defined in Table 138 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>1</td>
<td>0</td>
<td>1</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>lnl</td>
<td>Logically not less than</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</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 138.

<table>
<thead>
<tr>
<th>Trap Semantics</th>
<th>64-bit Comparison</th>
<th>32-bit Comparison</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>td 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>tdlti</td>
<td>tdtl</td>
</tr>
<tr>
<td>Trap if less than or equal</td>
<td>tdlei</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>tdliti</td>
<td>tdlit</td>
</tr>
<tr>
<td>Trap if not equal</td>
<td>tdnei</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>tdlli</td>
<td>tdllt</td>
</tr>
<tr>
<td>Trap if logically less than or equal</td>
<td>tdllei</td>
<td>tdlle</td>
</tr>
<tr>
<td>Trap if logically greater than or equal</td>
<td>tdgei</td>
<td>tdge</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>tdlnni</td>
<td>tdlnn</td>
</tr>
<tr>
<td>Trap if logically not greater than</td>
<td>tdlgni</td>
<td>tdlng</td>
</tr>
</tbody>
</table>
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 139, “Integer Select mnemonics,” on page 798 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 139.

<table>
<thead>
<tr>
<th>Table 139: Integer Select mnemonics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Select semantics</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.
   iselev 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>Table 140: Doubleword rotate and shift mnemonics</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Operation</strong></td>
</tr>
<tr>
<td>---</td>
</tr>
<tr>
<td>Extract and left justify immediate</td>
</tr>
<tr>
<td>Extract and right justify immediate</td>
</tr>
<tr>
<td>Insert from right immediate</td>
</tr>
<tr>
<td>Rotate left immediate</td>
</tr>
<tr>
<td>Rotate right immediate</td>
</tr>
<tr>
<td>Rotate left</td>
</tr>
<tr>
<td>Shift left immediate</td>
</tr>
<tr>
<td>Shift right immediate</td>
</tr>
<tr>
<td>Clear left immediate</td>
</tr>
<tr>
<td>Clear right immediate</td>
</tr>
<tr>
<td>Clear left and shift left immediate</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: `rldicr Rx,Rx,8,55`)

4. Clear the high-order 32 bits of register Ry and place the result into register Rx.
   
   `clrldi Rx,Ry,32` (equivalent to: `rldicl Rx,Ry,0,32`)

799  Power ISA™ I
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.

<table>
<thead>
<tr>
<th>Table 141: Word rotate and shift mnemonics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation</td>
</tr>
<tr>
<td>-----------------------------</td>
</tr>
<tr>
<td>Extract and left justify immediate</td>
</tr>
<tr>
<td>Extract and right justify immediate</td>
</tr>
<tr>
<td>Insert from left immediate</td>
</tr>
<tr>
<td>Insert from right immediate</td>
</tr>
<tr>
<td>Rotate left immediate</td>
</tr>
<tr>
<td>Rotate right immediate</td>
</tr>
<tr>
<td>Shift left immediate</td>
</tr>
<tr>
<td>Shift right immediate</td>
</tr>
<tr>
<td>Clear left immediate</td>
</tr>
<tr>
<td>Clear right immediate</td>
</tr>
<tr>
<td>Clear left and shift left immediate</td>
</tr>
</tbody>
</table>

Examples

1. Extract the sign bit (bit 32) of register Ry and place the result right-justified into register Rx.
   ```
   extwi Rx,Ry,1,0   (equivalent to: rlwinm Rx,Ry,1,31,31)
   ```

2. Insert the bit extracted in (1) into the sign bit (bit 32) of register Rz.
   ```
   insrwi Rz,Rx,1,0   (equivalent to: rlwimi Rz,Rx,31,0,0)
   ```

3. Shift the contents of register Rx left 8 bits, clearing the high-order 32 bits.
   ```
   slwi Rx,Rx,8   (equivalent to: rlwinm Rx,Rx,8,0,23)
   ```

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 Rx,Ry,0,16,31)
   ```
## 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 142: Extended mnemonics for moving to/from an SPR

<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>mtxer Rx</td>
<td>mt spr 1,Rx</td>
</tr>
<tr>
<td>DSCR</td>
<td>mtudscr Rx</td>
<td>mtspr 3,Rx</td>
</tr>
<tr>
<td>LR</td>
<td>mtlr Rx</td>
<td>mtspr 8,Rx</td>
</tr>
<tr>
<td>CTR</td>
<td>mtctr Rx</td>
<td>mtspr 9,Rx</td>
</tr>
<tr>
<td>AMR</td>
<td>mtuamr Rx</td>
<td>mtspr 13,Rx</td>
</tr>
<tr>
<td>TFHAR</td>
<td>mttfhar Rx</td>
<td>mtspr 128,Rx</td>
</tr>
<tr>
<td>TFIAR</td>
<td>mttfiar Rx</td>
<td>mtspr 129,Rx</td>
</tr>
<tr>
<td>TEXASR</td>
<td>mttxasr Rx</td>
<td>mtspr 130,Rx</td>
</tr>
<tr>
<td>TEXASRU</td>
<td>mttxasru Rx</td>
<td>mtspr 131,Rx</td>
</tr>
<tr>
<td>CTRL</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>VRSAVE</td>
<td>mtvrsave Rx</td>
<td>mtspr 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>MMCR2</td>
<td>mtumcr2 Rx</td>
<td>mtspr 769,Rx</td>
</tr>
<tr>
<td>MMCRA</td>
<td>mtumcr Ra</td>
<td>mtspr 770,Rx</td>
</tr>
<tr>
<td>PMC1</td>
<td>mtupmc1 Rx</td>
<td>mtspr 771,Rx</td>
</tr>
<tr>
<td>PMC2</td>
<td>mtupmc2 Rx</td>
<td>mtspr 772,Rx</td>
</tr>
<tr>
<td>PMC3</td>
<td>mtupmc3 Rx</td>
<td>mtspr 773,Rx</td>
</tr>
<tr>
<td>PMC4</td>
<td>mtupmc4 Rx</td>
<td>mtspr 774,Rx</td>
</tr>
<tr>
<td>PMC5</td>
<td>mtupmc5 Rx</td>
<td>mtspr 775,Rx</td>
</tr>
<tr>
<td>PMC6</td>
<td>mtupmc6 Rx</td>
<td>mtspr 776,Rx</td>
</tr>
<tr>
<td>MMCRO</td>
<td>mtumcmor Rx</td>
<td>mtspr 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>mtspr 800,Rx</td>
</tr>
<tr>
<td>BESCRU</td>
<td>mtbescru Rx</td>
<td>mtspr 801,Rx</td>
</tr>
<tr>
<td>BESCRR</td>
<td>mtbescrr Rx</td>
<td>mtspr 802,Rx</td>
</tr>
<tr>
<td>BESCRRR</td>
<td>mtbescr Ru</td>
<td>mtspr 803,Rx</td>
</tr>
<tr>
<td>EBBHR</td>
<td>mtebbhr Rx</td>
<td>mtspr 804,Rx</td>
</tr>
<tr>
<td>EBBRR</td>
<td>mtebbrr Rx</td>
<td>mtspr 805,Rx</td>
</tr>
<tr>
<td>BESCR</td>
<td>mtbescr Rx</td>
<td>mtspr 806,Rx</td>
</tr>
<tr>
<td>TAR</td>
<td>mttaar Rx</td>
<td>mtspr 815,Rx</td>
</tr>
<tr>
<td>PPR</td>
<td>mtppr Rx</td>
<td>mtspr 896,Rx</td>
</tr>
<tr>
<td>PPR32</td>
<td>mtppr32 Rx</td>
<td>mtspr 898,Rx</td>
</tr>
</tbody>
</table>
Examples

1. Copy the contents of register Rx to the XER.
   mtxer Rx
   (equivalent to: mtspr 1,Rx)

2. Copy the contents of the LR to register Rx.
   mflr Rx
   (equivalent to: mfspr Rx,8)

3. Copy the contents of register Rx to the CTR.
   mtctr Rx
   (equivalent to: mtspr 9,Rx)

C.10 Miscellaneous Mnemonics

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.

nop
   (equivalent to: ori 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 desirable. An extended mnemonic is provided for the executed form of no-op. This form of no-op will still consume execution resources.

xnop
   (equivalent to: xori 0,0,0)

Load Immediate

The addi and 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.
   li Rx,value
   (equivalent to: addi Rx,0,value)

Load a 16-bit signed immediate value, shifted left by 16 bits, into register Rx.
   lis Rx,value
   (equivalent to: addis Rx,0,value)

Load Next Instruction Address

The addpci instruction can be used to load the next instruction address into a register. An extended mnemonic is provided to perform this operation.

Inia Rx
   (equivalent to: addpci 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.

\[ \text{la } R_x, D(R_y) \quad (\text{equivalent to: } \text{addi } R_x, R_y, D) \]

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 \( R_x \).

\[ \text{la } R_x, v \quad (\text{equivalent to: } \text{addi } R_x, R_v, D_v) \]

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 \( R_y \) to register \( R_x \). This mnemonic can be coded with a final "\.*" to cause the Rc bit to be set in the underlying instruction.

\[ \text{mr } R_x, R_y \quad (\text{equivalent to: } \text{or } R_x, R_y, R_y) \]

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 \( R_y \) and places the result into register \( R_x \). This mnemonic can be coded with a final "\.*" to cause the Rc bit to be set in the underlying instruction.

\[ \text{not } R_x, R_y \quad (\text{equivalent to: } \text{nor } R_x, R_y, R_y) \]

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.

\[ \text{mtcr } R_x \quad (\text{equivalent to: } \text{mtcrf } 0xFF, R_x) \]

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.

\[ \text{mtcrf } FXM, R_x \quad \text{mfcr } R_x \]

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 system memory that holds supervisor software and general purpose applications and data, generally embodied as memory DIMMs attached to a memory controller which is in turn attached to the nest fabric. This is in contrast with memory associated with accelerators or I/O interfaces or attached to other systems

- **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.

- a transaction failure handler is invoked (see Section 5.3.3)
- an event-based branch occurs (see Chapter 7)
- the BHRB is read (see Section 8.2)

- **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 5.5 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.
- **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^n 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.
- **aggregate store**
  The set of stores caused by a successful transaction, which are performed as an atomic unit.

### 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 2^64-1. 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 simple 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 1.11.1 of Book I).

Quadword accesses with aligned storage operands are guaranteed to be atomic when caused by the following instructions.

- `iq`
- `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`
- `lfdp, lfdpx, 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 one quadword and is doubleword-aligned:
  - the access is performed as two disjoint aligned doubleword atomic accesses.
- 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 processor is the result stored by the first processor.
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 (icbi)
- 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 (dcbi, dcbtst)
- 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 dcbfi)

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
- Strong Access Order

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 ldax, ldaxr, ldaxr, ldaxr, ldarx, lqarx, stdcx, stdcx, stdcx, stdcx, and stdcx 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, 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.

Instructions are not fetched from virtual storage that is Guarded. If the instruction addressed by the current instruction address is in such storage, the system instruction storage error handler may be invoked (see Section 6.5.5 of Book III).

Programming Note

In some implementations, instructions may be executed before they are known to be required by the sequential execution model. Because the results of instructions executed in this manner are discarded if it is later determined that those instructions would not have been executed in the sequential execution model, this behavior does not affect most programs.

This behavior does affect programs that access storage locations that are not “well-behaved” (e.g., a storage location that represents a control register on an I/O device that, when accessed, causes the device to perform an operation). To avoid unintended results, programs that access such storage locations should request that the storage be Guarded, and should prevent such storage locations from being in a cache (e.g., by requesting that the storage also be Caching Inhibited).

1.6.5 Strong Access Order

All accesses to storage with the Strong Access Order (SAO) attribute (referred to as SAO storage) will be performed using a set of ordering rules different from that of the weakly consistent model that is described in Section 1.7.1, “Storage Access Ordering”. These rules apply only to accesses that are caused by a Load or a Store, and not to accesses associated with those instructions. Furthermore, these rules do not apply to accesses that are caused by or associated with instructions that are stated in their descriptions to be “treated as a Load” or “treated as a Store.” The details are described below, from the programmer’s point of view. (The processor may deviate from these rules if the programmer cannot detect the deviation.) The SAO attribute is not intended to be used for general purpose programming. It is provided in a manner that is not fully independent of the other storage attributes. Specifically, it is only provided for storage that is Memory Coherence Required, but not Write Through Required, not Caching Inhibited, and not Guarded. See Section 5.8.2.1, “Storage Control Bit Restrictions”, in Book III for more details. Accesses to SAO storage are likely to be performed more slowly than similar accesses to non-SA0 storage.

The order in which a processor performs storage accesses to SAO storage, the order in which those accesses are performed with respect to other processors and mechanisms, and the order in which those accesses are performed in main storage are the same except in the circumstances described in the following paragraph. The ordering rules for accesses performed by a single processor to SAO storage are as follows. Stores are performed in program order. When a store accesses data adjacent to that which is accessed by the next store in program order, the two storage accesses may be combined into a single larger access.

Loads are performed in program order. When a load accesses data adjacent to that which is accessed by the next load in program order, the two storage accesses may be combined into a single larger access. Stores may not be performed before loads which precede them in program order. Loads may be performed before stores which precede them in program order, with the provision that a load which follows a store of the same datum (to the same address) must obtain a value which is no older (in consideration of the possibility of programs on other processors sharing the same storage) than the value stored by the preceding store.

When any given processor loads the datum it just stored, as described above, the load may be performed by the processor before the preceding store has been performed with respect to other processors and mechanisms, and in main storage. This may cause the processor to see its store earlier relative to stores performed by other processors than it is observed by other processors and mechanisms, and than it is performed in memory. A direct consequence of this consideration is that although programs running on each processor will see the same sequence of accesses from any individual processor to SAO storage, each may in general see a different interleaving of the individual sequences. The memory barrier instructions may be used to establish stronger ordering, as described in Section 1.7.1, “Storage Access Ordering”, beginning with the third major bullet.

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 Power ISA defines two models for the ordering of storage accesses: weakly consistent and strong access ordering. The predominant model 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. Implementations which support SAO apply a stronger consistency model among accesses to SAO storage. The order between accesses to SAO storage and those performed using the weakly consistent model is characteristic of the weakly consistent model. The following description, through the second major bullet, applies only to the weakly consistent model. The corresponding description for SAO storage is found in Section 1.6.5, "Strong Access Order". The rest of the description following the second bulletted item applies to both models.

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. For each applicable pair ai, bj of storage accesses such that ai is in A and bj is in B, the memory barrier ensures that ai will be performed with respect to any processor or mechanism, to the extent required by the associated Memory Coherence Required attributes, before bj 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), even if the accesses are to SAO storage, 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 913.

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.
1.7.2 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.

1.7.3 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.

1.7.4 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 lwarx and stwcx; operation of the byte, halfword, doubleword, and quadword forms lbarx, stbcx, lharx, sthcx, ldarx, stdcx, ldarx, and stdcx, respectively, is the same except for obvious substitutions.

The 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 stwcx instruction is created.
2. The memory coherence mechanism is notified that a reservation exists for the storage location specified by the lwarx.

The stwcx instruction is a store to a word-aligned location that is conditioned on the existence of the reservation created by the 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 lwarx and the stwcx specify the same storage location.

A stwcx performs a store to the target storage location only if the reservation created by the lwarx still exists at the time the 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 implementa-
tion. 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 Store Conditional instruction is used with a preceding Load And Reserve instruction that has a different storage operand length (e.g., stwcx. with ldarx), whether the store is performed is undefined. Otherwise the store is performed.

A stwcx. that performs its store is said to "succeed".

Examples of the use of lwarx and stwcx. are given in Appendix B. "Programming Examples for Sharing Storage" on page 913.

A successful 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 lwarx from the given location by another processor may return a "stale" value. However, a subsequent lwarx from the given location by the other processor followed by a successful stwcx. by that processor is guaranteed to have returned the value stored by the first processor’s stwcx. (in the absence of other stores to the given location).

--- Programming Note ---

The store caused by a successful stwcx. is ordered, by a dependence on the reservation, with respect to the load caused by the lwarx that established the reservation, such that the two storage accesses are performed in program order with respect to any processor or mechanism.

1.7.4.1 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 lqarx 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 1-9 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, lharx, lwarx, 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, lharx, lwarx, ldarx, or lqarx that established the reservation, and regardless of whether the storage operand 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. Any of the following occurs on the processor holding the reservation.
   a. The transaction state changes (from Non-transactional, Transactional, or Suspended state to one of the other two states; see Section 5.2, "Transactional Memory Facility States"), except in the following cases
      ▪ If the change is from Transactional state to Suspended state, the reservation is not lost.
      ▪ If the change is from Suspended state to Transactional state, the reservation is not lost if it was established in Transactional state.
      ▪ If the change is caused by a treclaim, or trechpt, instruction, whether the reservation is lost is undefined.
   b. The transaction nesting depth (see Section 5.4, "Transactional Memory Facility Registers") changes; whether the reservation is lost is undefined. (This item applies only if the processor is in Transactional state both before and after the change.)
   c. The processor is in Suspended state and executes a Store Conditional instruction (stbcx, sthcx, stwcx, stdcx, or stqcx) or a waitrsv instruction; the reservation is lost if it was established in Transactional state. In this case the Store Conditional instruction's store is not performed, and the waitrsv does not wait. (For Store Conditional, the reservation is also lost if it was established in Suspended state; see item 2.)

5. Some other processor executes a Store or dcbz that specifies a location in the same reservation granule.

6. Some other processor executes a dcbst, or dcbt that specifies a location in the same reservation granule: whether the reservation is lost is undefined. (For a dcbst instruction that specifies a data stream, "location" in the preceding sentence includes all locations in the data stream.)

7. Any processor modifies a Reference or Change bit in the same reservation granule: whether the reservation is lost is undefined.

8. Some mechanism other than a processor modifies a storage location in the same reservation granule.

9. 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.

10. Implementation-specific characteristics of the coherence mechanism cause the reservation to be lost.

---

**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
- Software accesses a Device Control Register
1.7.4.2 Forward Progress

Forward progress in loops that use \texttt{lwarx} and \texttt{stwcx} is achieved by a cooperative effort among hardware, system software, and application software.

The architecture guarantees that when a processor executes a \texttt{lwarx} to obtain a reservation for location \( X \) and then a \texttt{stwcx} to store a value to location \( X \), either

1. the \texttt{stwcx} succeeds and the value is written to location \( X \), or
2. the \texttt{stwcx} fails because some other processor or mechanism modified location \( X \), or
3. the \texttt{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 Section 1.7.4.1.

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.

---

Programming Note

One use of \texttt{lwarx} and \texttt{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 913. 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 \texttt{lwarx} and \texttt{stwcx} improves on such a Compare and Swap, because the reservation reliably binds the \texttt{lwarx} and \texttt{stwcx} together. The reservation is always lost if the word is modified by another processor or mechanism between the \texttt{lwarx} and \texttt{stwcx}, so the \texttt{stwcx} never succeeds unless the word has not been stored into (by another processor or mechanism) since the \texttt{lwarx}.

Programming Note

In general, programming conventions must ensure that \texttt{lwarx} and \texttt{stwcx} specify addresses that match; a \texttt{stwcx} should be paired with a specific \texttt{lwarx} to the same storage location. Situations in which a \texttt{stwcx} may erroneously be issued after some \texttt{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 \texttt{lwarx} and before the paired \texttt{stwcx}. The \texttt{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 \texttt{stbxc}, \texttt{sthcx}, \texttt{stwcx}, \texttt{stdcx}, or \texttt{stqcx} that specifies a dummy writable aligned location as part of the context switch; see Section 6.4.3 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 \texttt{lwarx} and \texttt{stwcx} (e.g., pointers in certain linked lists; see Section B.3, “List Insertion” on page 917).
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 Transactions

A transaction is a group of instructions that collectively have unique storage access behavior intended to facilitate parallel programming. (It is possible to nest transactions within one another. The description in this chapter will ignore nesting because it does not have a significant impact on the properties of the memory model. Nesting and its consequences will be described elsewhere.) Sequences of instructions that are part of the transaction may be interleaved with sequences of Suspended state instructions that are not part of the transaction. A transaction is said to “succeed” or to “fail,” and failure may happen before all of the instructions in the transaction have completed. If the transaction fails, it is as if the instructions that are part of the transaction were never executed. If the transaction succeeds, it appears to execute as an atomic unit as viewed by other processors and mechanisms. (Although the transaction appears to execute atomically, some knowledge of the inner workings will be necessary to avoid apparent paradoxes in the rest of the model. These details are described below.) The execution of Suspended state sequences have the same effect that the sequence would have in the absence of a transaction, independent of the success or failure of the transaction, including accessing storage according to the weakly consistent storage model or SAO, based on storage attributes. Upon failure, normal execution continues at the failure handler. Except for the rollback of the effects of transactional instructions upon transaction failure, as viewed by the executing thread, the interleaved sequences of Transactional and Suspended state instructions appear to execute according to the sequential execution model. See Chapter 5 “Transactional Memory Facility” on page 877 for more details. The unique attributes of the storage model for transactions are described below.

Transaction processing does not support the rollback of operations on the reservation mechanism. To prevent this possibility, a reservation is lost as a result of a state change from Transactional to Non-transactional or Non-transactional to Transactional. It is possible to successfully complete an atomic update in Transactional state, though such a sequence would have no benefit. It is also possible to complete an atomic update in Suspended state, or straddling an interval in Suspended state if Suspended state is entered via an interrupt or `tsuspend`, and exited via `tresume`, `rfebb`, `rfid`, `rfscv`, `hrfid`, or `mtmsrd`. However, an atomic update will not succeed if only one of the `Load and Reserve` / `Store Conditional` instruction pair is executed in Suspended state.

Programming Note

Note that if a `Store Conditional` instruction within a transaction does not store, it may still be possible for the transaction to succeed. Software must not depend on the two operations having the same outcome. For example, software must not use success of an enclosing transaction as a replacement for checking the condition code from a transactional `Store Conditional` instruction.

Programming Note

Accessing storage locations in Suspended state that have been accessed transactionally has the potential to create apparent storage paradoxes. Consider, for example, a case where variable X has initial value zero, is updated transactionally to one, is read in Suspended state, subsequently the transaction fails, and variable X is read again. In the absence of external conflicts, the observed sequence of values will be zero, one, zero: old, new, old.

Performing an atomic update on X in Suspended state may be even more confusing. Suppose the atomic sequence increments X, but that the only way to have X=1 is via the transactional store that occurs before entering Suspended state. The store conditional, if it succeeds, will store X=2 and in so doing, kill the transaction. But with the transaction having failed, X was never equal to one.

The flexibility of the Suspended state programming model can create unintuitive results. It must be used with care.

Successful transactions are serialized in some order, and no processor or mechanism is able to observe the accesses caused by any subset of these transactions as occurring in an order that conflicts with this order. Specifically, let processor i execute transactions 0, 1, ..., j, ..., k, j*1, ..., where only successful transactions are numbered, and the numbering reflects program order. Let Tij be transaction j on processor i. Then there is an ordering of the Tij such that no processor or mechanism is able to observe the accesses caused by the transactions Tij in an order that conflicts with this ordering. Note that Suspended state storage accesses are not included in the serialization property.
Because of the difference between a transaction's instantaneous appearance and the finite time required to execute it in an implementation, it is exposed to changes in memory management state in a way that is not true for individual accesses. A change to the translation or protection state that would prevent any access from taking place at any time during its processing for the transaction compromises the integrity of the transaction. Any such change must either be prevented or must cause the transaction to fail. The architecture will automatically fail a transaction if the memory management state change is accomplished using tlbie or slbieg. An implementation may overdetect such conflicts between the tlbie or slbieg and the transaction footprint. (Overdetection may result from the technique used to detect the conflict. A bloom filter may be used, as an example. Subsequent references to translation invalidation conflicts implicitly include any cases of spurious overdetection.) Changes made in some other manner must be managed by software, for example by explicitly terminating any affected transactions. Examples of instructions that require software management are tlbie, slbie, slbia, and sliag.

The atomic nature of a transaction, together with the cumulative memory barrier created by the transaction and the memory barriers created by tbegin, and tend, described below, has the potential to eliminate the need for explicit memory barriers within the transaction, and before and after the transaction as well. However, since there may be a desire to preserve existing algorithms while exploiting transactions, the interaction of memory barriers and transactions is defined. In the presence of transactions, storage access ordering is the same as if no transactions are present, with the following exceptions. Memory barriers that are created while the transaction is running (other than the integrated cumulative memory barrier of the transaction described below), data dependencies, and SAO do not order transactional stores. Instead, transactional stores are grouped together into an “aggregate store,” which is performed as an atomic unit with respect to other processors and mechanisms when the transaction succeeds, after all the transactional loads have been performed. With this store behavior, the appearance of transactional atomicity is created in a manner similarly to that for a Load and Reserve / Store Conditional pair. Success of the transaction is conditional on the storage locations specified by the loads not having been stored into by a more recent Suspended state store or by any store by another processor or mechanism since the load was performed. (There are additional conditions for the success of transactions.)

A tbegin instruction that begins a successful transaction creates a memory barrier that immediately precedes the transaction and orders storage accesses pairwise, as follows. Let A and B be sets of storage accesses as defined below. For each pair aibi of storage accesses such that ai is in A and bi is in B, the memory barrier ensures that ai will be performed with respect to any processor or mechanism, to the extent required by the associated Memory Coherence Required attributes, before bi is performed with respect to that processor or mechanism. Set A contains all data accesses caused by instructions preceding the tbegin, that are neither Write Through Required nor Caching Inhibited. Set B contains all data accesses caused by instructions following the tbegin, including Suspended state accesses, that are neither Write Through Required nor Caching Inhibited. The ordering done by this memory barrier is cumulative.

A successful transaction has an integrated cumulative memory barrier behavior. When a processor (P1) executes a tend. instruction and tend. processing determines that the transaction will succeed, a memory barrier is created, which orders storage accesses pairwise, as follows. Let A and B be sets of storage accesses as defined below. For each pair aibi of storage accesses such that ai is in A and bi is in B, the memory barrier ensures that ai will be performed with respect to any processor or mechanism, to the extent required by the associated Memory Coherence Required attributes, before bi is performed with respect to that processor or mechanism. Set A contains all non-transactional data accesses by other processors and mechanisms that have been performed with respect to P1 before the memory barrier is created and are neither Write Through Required nor Caching Inhibited. Set B contains the aggregate store and all non-transactional data accesses by other processors and mechanisms that are performed after a Load instruction executed by that processor or mechanism has returned the value stored by a store that is in set B.

Note that the integrated cumulative memory barrier does not order Suspended state storage accesses interleaved with the transaction.

A tend. instruction that ends a successful transaction creates a memory barrier that immediately follows the transaction and orders storage accesses pairwise, as follows. Let A and B be sets of storage accesses as defined below. For each pair aibi of storage accesses such that ai is in A and bi is in B, the memory barrier ensures that ai will be performed with respect to any processor or mechanism, to the extent required by the associated Memory Coherence Required attributes, before bi is performed with respect to that processor or
mechanism. Set A contains all data accesses caused by instructions preceding the \texttt{tend}, including Suspended state accesses, that are neither Write Through Required nor Caching Inhibited. Set B contains all data accesses caused by instructions following the \texttt{tend} that are neither Write Through Required nor Caching Inhibited. The ordering done by this memory barrier is cumulative.

\begin{table}[h]
\centering
\begin{tabular}{|l|}
\hline
Programming Note
\hline
The memory barriers that are created by the execution of a successful transaction (those associated with \texttt{tbegin}, \texttt{tend}, and the integrated cumulative memory barrier) render most explicit memory barriers in and around transactions redundant. An exception is when there is a need to establish order among Suspended state accesses.
\hline
\end{tabular}
\end{table}

1.8.1 Rollback-Only Transactions
A Rollback-Only Transaction (ROT) is a sequence of instructions that is executed, or not, as a unit. The purpose of the ROT is to enable bulk speculation of instructions with minimum overhead. It leverages the rollback mechanism that is invoked as part of transaction failure handling, but has reduced overhead in that it does not have the full atomic nature of the transaction and its synchronization and serialization properties. The absence of a (normal) transaction’s atomic quality means that a ROT must not be used to manipulate shared data.

More specifically, a ROT differs from a normal transaction as follows.
- ROTs are not serialized.
- There are no memory barriers created by \texttt{tbegin}. and \texttt{tend}.
- A ROT has no integrated cumulative memory barrier.
- There is no monitoring of storage locations specified by loads for modification by other processors and mechanisms between the performing of the loads and the completion of the ROT.
- The stores that are included in the ROT need not appear to be performed as an aggregate store. (Implementations are likely to provide an aggregate store appearance, but the correctness of the program must not depend on the aggregate store appearance.)

1.9 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).

\begin{table}[h]
\centering
\begin{tabular}{|l|}
\hline
Programming Note
\hline
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.
\hline
\end{tabular}
\end{table}

For each instance of executing an instruction from location X, the instruction may be fetched multiple times. 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 1.9.1, “Concurrent Modification and Execution of Instructions” on page 825.
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`.

```
li r0,1  # put a 1 value in r0
dcbst X  # copy the block in main storage
sync    # order copy before invalidation
icbi X  # invalidate copy in instr cache
sync    # 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
            cmpwi r0,1 #   loaded, location X in inst'n
            bne $-8    #   storage is consistent with
            #   location X in data storage)
isync    # discard any prefetched inst'ns
```

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
sync    # order copy before invalidation
icbi X  # invalidate copy in instr cache
isync    # discard any prefetched inst'ns
```

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.
1.9.1 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 is not consistent with data storage or which becomes inconsistent with data storage prior to the completion of its processing. 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 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.
- \( X_1, X_2, ..., X_n \) 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 instructions consists of the I-form \( \text{Branch} \) instruction (\( b[l][a] \)) and the preferred no-op instruction (\( \text{ori} 0,0,0 \)).

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 \( X_1, ..., X_n \) into location X are atomic stores that modify all four bytes of location X.
2. Each \( X_i, 0 \leq i \leq n \), is a patch class instruction.
3. Location X is in storage that is Memory Coherence Required.

If these conditions are satisfied, the result of each execution of an instruction from location X will be the execution of 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 at location X could be the sequence \( X_i, X_{i+2}, X_{i+1} \)). 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 \textit{Add Immediate} instruction, it will appear that a legal instruction caused an illegal instruction exception.

Programming Note
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 1.9 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 a patch class instruction in the original program with a \textit{Branch} instruction that would cause any program executing the \textit{Branch} to branch to the newly created code. The first instruction in the newly created code must be an \textit{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 \textit{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 \textit{Store} which stores the new \textit{Branch} instruction is performed.

Programming Note
It is believed that all processors that comply with versions of the architecture that precede Version 2.01 support concurrent modification and execution of instructions as described in this section if the requirements given above are satisfied, and that most such processors yield boundedly undefined results if the requirements given above are not satisfied. However, in general such support has not been verified by processor testing. Also, one such processor is known to yield undefined results in certain cases if the requirements given above are not satisfied.
Chapter 2. Performance Considerations and Instruction Restart

2.1 Performance-Optimized Instruction Sequences

Performance-optimized instruction sequences are instruction sequences that provide better performance than other ways of achieving the same results. The supported performance-optimized sequences are shown in the following sections. In order to achieve the improved performance, the sequences must be coded exactly as shown, including instruction order, register re-use, and lack of intervening instructions. The processor achieves the improved performance by executing the sequence as a single operation, or in some other highly efficient, sequence-specific, manner. (The improved performance may not be obtained if the sequence causes the system error handler to be invoked, or for implementation-dependent reasons.)
### 2.1.1 Load and Store Operations

The following instruction sequences will optimize performance for storage accesses to effective addresses that are offset from (RA) by magnitudes of up to $2^{32}$.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Load Instruction Sequence</th>
<th>Store Instruction Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fixed-point byte accesses</td>
<td>addis Rx,RA,SI</td>
<td>addis Rx,RA,SIh</td>
</tr>
<tr>
<td></td>
<td>lbz           Rs,D(Rx)</td>
<td>stb RS,D(Rx)</td>
</tr>
<tr>
<td>Fixed-point halfword accesses</td>
<td>addis Rx,RA,SIh</td>
<td>addis Rx,RA,SIh</td>
</tr>
<tr>
<td></td>
<td>lhz           Rs,D(Rx)</td>
<td>sth RS,D(Rx)</td>
</tr>
<tr>
<td>Fixed-point word accesses</td>
<td>addis Rx,RA,SIh</td>
<td>addis Rx,RA,SIh</td>
</tr>
<tr>
<td></td>
<td>lwz           Rs,D(Rx)</td>
<td>stw RS,D(Rx)</td>
</tr>
<tr>
<td>Fixed-point doubleword accesses</td>
<td>addis Rx,RA,SIh</td>
<td>addis Rx,RA,SIh</td>
</tr>
<tr>
<td></td>
<td>ld            Rs,D(Rx)</td>
<td>std RS,D(Rx)</td>
</tr>
<tr>
<td>Floating-point single-precision accesses</td>
<td>addis Rx,RA,SIh</td>
<td>addis Rx,RA,SIh</td>
</tr>
<tr>
<td></td>
<td>lfs            Rs,D(Rx)</td>
<td>std RS,D(Rx)</td>
</tr>
<tr>
<td>Floating-point double-precision accesses</td>
<td>addis Rx,RA,SIh</td>
<td>addis Rx,RA,SIh</td>
</tr>
<tr>
<td></td>
<td>lfd            Rs,D(Rx)</td>
<td>stdf RS,D(Rx)</td>
</tr>
<tr>
<td>VSX Scalar doubleword accesses</td>
<td>addis Rx,RA,SIh</td>
<td>addis Rx,RA,SIh</td>
</tr>
<tr>
<td></td>
<td>lxsd           XT,DS(Rx)</td>
<td>stxsd XS,DS(Rx)</td>
</tr>
<tr>
<td>VSX Scalar single-precision accesses</td>
<td>addis Rx,RA,SIh</td>
<td>addis Rx,RA,SIh</td>
</tr>
<tr>
<td></td>
<td>lxssp          XT,DS(Rx)</td>
<td>stxssp XS,DS(Rx)</td>
</tr>
<tr>
<td>VSX Vector accesses</td>
<td>addis Rx,RA,SIh</td>
<td>addis Rx,RA,SIh</td>
</tr>
<tr>
<td></td>
<td>lxv            XT,DQ(Rx)</td>
<td>stxv XS,DQ(Rx)</td>
</tr>
</tbody>
</table>

Table 1: Loads and Stores with offsets of up to $2^{32}$ offsets from base register.
The following instruction sequences will optimize performance for storage accesses to effective addresses that are offset from (RB) by magnitudes of up to $2^{16}$.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Load Instruction Sequence</th>
<th>Store Instruction Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fixed-point doubleword accesses</td>
<td><code>addi Rx,0,SI</code></td>
<td><code>addi Rx,0,SI</code></td>
</tr>
<tr>
<td></td>
<td><code>ldx Rt,RA,Rx</code></td>
<td><code>stdx RS,RA,Rx</code></td>
</tr>
<tr>
<td>Floating-point as integer word accesses</td>
<td><code>addi Rx,0,SI</code></td>
<td><code>addi Rx,0,SI</code></td>
</tr>
<tr>
<td></td>
<td><code>lfiwz x FRT,RA,Rx</code></td>
<td><code>stfiwx FRS,RA,Rx</code></td>
</tr>
<tr>
<td>Vector byte accesses</td>
<td><code>addi Rx,0,SI</code></td>
<td><code>addi Rx,0,SI</code></td>
</tr>
<tr>
<td></td>
<td><code>lvebx VRT,RA,Rx</code></td>
<td><code>stvebx VRS,RA,Rx</code></td>
</tr>
<tr>
<td>Vector halfword accesses</td>
<td><code>addi Rx,0,SI</code></td>
<td><code>addi Rx,0,SI</code></td>
</tr>
<tr>
<td></td>
<td><code>lvehx VRT,RA,Rx</code></td>
<td><code>stvehx VRS,RA,Rx</code></td>
</tr>
<tr>
<td>Vector word accesses</td>
<td><code>addi Rx,0,SI</code></td>
<td><code>addi Rx,0,SI</code></td>
</tr>
<tr>
<td></td>
<td><code>lvewx VRT,RA,Rx</code></td>
<td><code>stvewx VRS,RA,Rx</code></td>
</tr>
<tr>
<td>Vector accesses</td>
<td><code>addi Rx,0,SI</code></td>
<td><code>addi Rx,0,SI</code></td>
</tr>
<tr>
<td></td>
<td><code>lvx VRT,RA,Rx</code></td>
<td><code>stvx VRS,RA,Rx</code></td>
</tr>
<tr>
<td>VSX Vector accesses</td>
<td><code>addi Rx,0,SI</code></td>
<td><code>addi Rx,0,SI</code></td>
</tr>
<tr>
<td></td>
<td><code>lxv x XT,RA,Rx</code></td>
<td><code>stvx XRS,RA,Rx</code></td>
</tr>
<tr>
<td>VSX Vector doubleword accesses</td>
<td><code>addi Rx,0,SI</code></td>
<td><code>addi Rx,0,SI</code></td>
</tr>
<tr>
<td></td>
<td><code>lxvd2x XT,RA,Rx</code></td>
<td><code>stxvd2x XRS,RA,Rx</code></td>
</tr>
<tr>
<td>VSX Vector word accesses</td>
<td><code>addi Rx,0,SI</code></td>
<td><code>addi Rx,0,SI</code></td>
</tr>
<tr>
<td></td>
<td><code>lxvw4x XT,RA,Rx</code></td>
<td><code>stxvw4x XRS,RA,Rx</code></td>
</tr>
<tr>
<td>VSX Vector halfword accesses</td>
<td><code>addi Rx,0,SI</code></td>
<td><code>addi Rx,0,SI</code></td>
</tr>
<tr>
<td></td>
<td><code>lxvh8x XT,RA,Rx</code></td>
<td><code>stxvh8x XRS,RA,Rx</code></td>
</tr>
<tr>
<td>VSX Vector byte accesses</td>
<td><code>addi Rx,0,SI</code></td>
<td><code>addi Rx,0,SI</code></td>
</tr>
<tr>
<td></td>
<td><code>lxvb16x XT,RA,Rx</code></td>
<td><code>stxvb16x XRS,RA,Rx</code></td>
</tr>
<tr>
<td>VSX Vector word splat accesses</td>
<td><code>addi Rx,0,SI</code></td>
<td><code>n/a</code></td>
</tr>
<tr>
<td></td>
<td><code>lxwsx XT,RA,Rx</code></td>
<td><code>n/a</code></td>
</tr>
<tr>
<td>VSX Vector doubleword splat accesses</td>
<td><code>addi Rx,0,SI</code></td>
<td><code>n/a</code></td>
</tr>
<tr>
<td></td>
<td><code>lxvsx XT,RA,Rx</code></td>
<td><code>n/a</code></td>
</tr>
<tr>
<td>VSX Scalar doubleword accesses</td>
<td><code>addi Rx,0,SI</code></td>
<td><code>addi Rx,0,SI</code></td>
</tr>
<tr>
<td></td>
<td><code>lxssdx XT,RA,Rx</code></td>
<td><code>stxsdx XRS,RA,Rx</code></td>
</tr>
<tr>
<td>VSX Scalar single-precision accesses</td>
<td><code>addi Rx,0,SI</code></td>
<td><code>addi Rx,0,SI</code></td>
</tr>
<tr>
<td></td>
<td><code>lxsspx XT,RA,Rx</code></td>
<td><code>stxspx XRS,RA,Rx</code></td>
</tr>
<tr>
<td>VSX Scalar byte accesses</td>
<td><code>addi Rx,0,SI</code></td>
<td><code>addi Rx,0,SI</code></td>
</tr>
<tr>
<td></td>
<td><code>lxssibzx XT,RA,Rx</code></td>
<td><code>stxsibx XRS,RA,Rx</code></td>
</tr>
<tr>
<td>VSX Scalar halfword accesses</td>
<td><code>addi Rx,0,SI</code></td>
<td><code>addi Rx,0,SI</code></td>
</tr>
<tr>
<td></td>
<td><code>lxssihzx XT,RA,Rx</code></td>
<td><code>stxsihx XRS,RA,Rx</code></td>
</tr>
<tr>
<td>VSX Scalar word accesses</td>
<td><code>addi Rx,0,SI</code></td>
<td><code>addi Rx,0,SI</code></td>
</tr>
<tr>
<td></td>
<td><code>lxssiwzx XT,RA,Rx</code></td>
<td><code>stxsiwx XRS,RA,Rx</code></td>
</tr>
</tbody>
</table>

Table 2: Loads and Stores with Offsets from (RA) by Magnitudes of Up to $2^{16}$. 
Programming Note

Even independent of the performance optimization described above, the techniques illustrated in Table 1 and Table 2 generally perform better than other ways of achieving the effect of having a large displacement field for D-form and DS-form fixed-point Load/Store instructions (Table 1), and of having a displacement field for X-form Vector and VSX Load/Store instructions (Table 2).

The technique for the fixed-point Load/Store instructions is complicated by the fact that D-form and DS-form Loads and Stores treat the D/DS value as signed.

For simplicity, most of this Note assumes that the fixed-point Load/Store instruction is D-form; the modifications for DS-form fixed-point Load/Store instructions are straightforward.

Let the desired effective address to load from or store to be (RA) + DISP, where DISP is a signed 32-bit value.

\[(RA) + DISP = (RA) + \text{DISP}_{0:15} \ || \ \text{DISP}_{16:31}\]

where \(\text{DISP}_{0:15}\) is a signed 16-bit value.

If \(\text{DISP}_{0:15}\) is used as the SI value for the \textit{addis}, the \textit{addis} forms the sum

\[(RA) + (\text{DISP}_{0:15} || 0x0000)\]

and places the result into Rx.

If \(\text{DISP}_{16:31}\) is used as the D value for the Load or Store and Rx is used as the base register for the Load or Store, and \(\text{DISP}_{16} = 0\), the Load or Store computes the EA to load from as

\[(Rx) + \text{DISP}_{16:31} = (RA) + (\text{DISP}_{0:15} || 0x0000) + \text{DISP}_{16:31}\]

\[= (RA) + \text{DISP}\]

However, because D-form Loads and Stores treat the D value as signed, if \(\text{DISP}_{16} = 1\) the Load or Store computes the EA as

\[\text{To compensate for this effective subtraction of } 2^{16}, \text{ if } \text{DISP}_{16} = 1 \text{ the } \text{addis} \text{ must be } \text{DISP}_{0:15} = 1. \text{ Then the } \text{addis} \text{ sets Rx to}\]

\[(RA) + ((\text{DISP}_{0:15} + 1) || 0x0000) = (RA) + (\text{DISP}_{0:15} || 0x0000) + 2^{16}\]

and the Load or Store computes the EA as

\[(Rx) + \text{DISP}_{16:31} = (RA) + (\text{DISP}_{0:15} || 0x0000) + 2^{16} + \text{DISP}_{16:31} - 2^{16}\]

\[= (RA) + \text{DISP}\]

as desired.

Thus the rules for using the technique illustrated in Table 1 are as follows.

- For the RA field of the \textit{addis}, use the desired base register for the Load or Store.
- For the D field of the Load or Store, use \text{DISP}_{16:31}.
- (For DS-form Loads and Stores, for the DS field use \text{DISP}_{16:29}; \text{DISP}_{30:31} are 0b00.)
- For the SI field of the \textit{addis}:
  - if \(\text{DISP}_{16} = 0\) use \text{DISP}_{0:15};
  - if \(\text{DISP}_{16} = 1\) use \text{DISP}_{0:15} + 1.
2.1.2 32-Bit Constant Generation

The following instruction sequences will optimize performance when generating zero-extended 32-bit unsigned constants (when RA_{0:63} equal 0) and when performing 32-bit logical operations on RA_{32:63}.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Instruction Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsigned constant (UI_h,UI_l zero extended)</td>
<td>oris Rx,RA,UI_h</td>
</tr>
<tr>
<td></td>
<td>ori Rx,RA,UI_l</td>
</tr>
<tr>
<td>Unsigned constant (UI_h,UI_l zero extended)</td>
<td>xoris Rx,RA,UI_h</td>
</tr>
<tr>
<td></td>
<td>xori Rx,RA,UI_l</td>
</tr>
</tbody>
</table>

Table 3: 32-bit Unsigned Constant Generation

The following instruction sequences will optimize performance when generating 32-bit signed constants.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Instruction Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signed constant (SI_h,SI_l sign extended)</td>
<td>addis Rx,RA,SI_h</td>
</tr>
<tr>
<td></td>
<td>addi Rx,RA,SI_l</td>
</tr>
<tr>
<td>Signed constant (SI_h sign extended; UI zero extended)</td>
<td>addis Rx,0,SI_h</td>
</tr>
<tr>
<td></td>
<td>ori Rx,RA,UI_l</td>
</tr>
</tbody>
</table>

Table 4: 32-bit Signed Constant Generation

2.1.3 Sign and Zero Extension

The following instruction sequences will optimize performance when zero-extending the result of a 32-bit addition.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Instruction Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsigned constant (RA + RB zero extended)</td>
<td>add Rx,RA,RB</td>
</tr>
<tr>
<td></td>
<td>ridcl Rt,Rx,0,32</td>
</tr>
</tbody>
</table>

Table 6: 32-bit Zero-Extended Addition
2.1.4 Load/Store Addressing Relative to Program Counter

The following instruction sequences will optimize performance for storage accesses to effective addresses that are offset from the CIA by magnitudes of up to $2^{32}$.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Load Instruction Sequence</th>
<th>Store Instruction Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fixed-point byte accesses</td>
<td>addpcis Rx,SIh</td>
<td>addpcis Rs,SIh</td>
</tr>
<tr>
<td></td>
<td>lbz           Rs,DI(Rx)</td>
<td>stb Rs,DI(Rx)</td>
</tr>
<tr>
<td>Fixed-point halfword accesses</td>
<td>addpcis Rx,SIh</td>
<td>addpcis Rs,SIh</td>
</tr>
<tr>
<td></td>
<td>lhz           Rs,DI(Rx)</td>
<td>sth Rs,DI(Rx)</td>
</tr>
<tr>
<td>Fixed-point word accesses</td>
<td>addpcis Rx,SIh</td>
<td>addpcis Rs,SIh</td>
</tr>
<tr>
<td></td>
<td>lwz           Rs,DI(Rx)</td>
<td>stw Rs,DI(Rx)</td>
</tr>
<tr>
<td>Fixed-point doubleword accesses</td>
<td>addpcis Rx,SIh</td>
<td>addpcis Rs,SIh</td>
</tr>
<tr>
<td></td>
<td>ld             Rs,DI(Rx)</td>
<td>std Rs,DI(Rx)</td>
</tr>
<tr>
<td>Fixed-point doubleword accesses</td>
<td>addpcis Rx,SIh</td>
<td>addpcis Rs,SIh</td>
</tr>
<tr>
<td></td>
<td>ldx            Rs,DI(Rx)</td>
<td>stdx Rs,DI(Rx)</td>
</tr>
<tr>
<td>Floating-point single-precision accesses</td>
<td>addpcis Rx,SIh</td>
<td>addpcis Rs,SIh</td>
</tr>
<tr>
<td></td>
<td>lfs             Rs,DI(Rx)</td>
<td>stfs Rs,DI(Rx)</td>
</tr>
<tr>
<td>Floating-point double-precision accesses</td>
<td>addpcis Rx,SIh</td>
<td>addpcis Rs,SIh</td>
</tr>
<tr>
<td></td>
<td>lfd             Rs,DI(Rx)</td>
<td>stfd Rs,DI(Rx)</td>
</tr>
<tr>
<td>VSX Scalar doubleword accesses</td>
<td>addpcis Rx,SIh</td>
<td>addpcis Rs,SIh</td>
</tr>
<tr>
<td></td>
<td>lxsds           Rs,DI(Rx)</td>
<td>stxsd Rs,DI(Rx)</td>
</tr>
<tr>
<td>VSX Scalar single-precision accesses</td>
<td>addpcis Rx,SIh</td>
<td>addpcis Rs,SIh</td>
</tr>
<tr>
<td></td>
<td>lxxsp           Rs,DI(Rx)</td>
<td>stxxsp Rs,DI(Rx)</td>
</tr>
<tr>
<td>VSX Vector accesses</td>
<td>addpcis Rx,SIh</td>
<td>addpcis Rs,SIh</td>
</tr>
<tr>
<td></td>
<td>lxv             Rs,DI(Rx)</td>
<td>stxv Rs,DI(Rx)</td>
</tr>
</tbody>
</table>

Table 7: Fixed-Point, Floating-Point and VSX Load/Store Fusion with offset up to $2^{32}$ from Program Counter

Programming Note

See the Programming Notes for Table 1.
2.1.5 Destructive Operation Operand Preservation

A destructive operation is an operation that modifies one of its inputs. The VSX Vector Permute and VSX Vector Multiply-Add instructions are destructive operations because they use their destination register as a source register.

When there is a need to preserve the contents of the overwritten source register for the various VSX Vector Permute and VSX Vector Multiply-Add instructions, performance will be optimized if the \texttt{xxlor} instruction is used to copy the contents of the source operand into another register, and then that register is used as the destination (and source) register for the VSX Vector Permute or VSX Vector Multiply-Add instruction.

As an example, to preserve the XT source register in the \texttt{xxperm} instruction, the following sequence will optimize performance.

\begin{verbatim}
xxlor XT,XC,XC /* Copy (XC) to XT
xxperm XT,XA,XB /* Permute, overwriting XT
\end{verbatim}

The set of instructions listed below, when immediately preceded by the \texttt{xxlor} XT,XC,XC instruction in a sequence similar to the above example, will provide optimal performance.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{xxperm}</td>
<td>XT,XA,XB VSX Vector Permute</td>
</tr>
<tr>
<td>\texttt{xxpermr}</td>
<td>XT,XA,XB VSX Vector Permute Right Indexed</td>
</tr>
<tr>
<td>\texttt{xsmaddasp}</td>
<td>XT,XA,XB VSX Scalar Multiply-Add Type-A Single-Precision</td>
</tr>
<tr>
<td>\texttt{xsmsubasp}</td>
<td>XT,XA,XB VSX Scalar Multiply-Subtract Type-A Single-Precision</td>
</tr>
<tr>
<td>\texttt{xsnmaddasp}</td>
<td>XT,XA,XB VSX Scalar Negative Multiply-Add Type-A Single-Precision</td>
</tr>
<tr>
<td>\texttt{xsnmsubasp}</td>
<td>XT,XA,XB VSX Scalar Negative Multiply-Subtract Type-A Single-Precision</td>
</tr>
<tr>
<td>\texttt{xsmaddadp}</td>
<td>XT,XA,XB VSX Scalar Multiply-Add Type-A Double-Precision</td>
</tr>
<tr>
<td>\texttt{xsmsubadp}</td>
<td>XT,XA,XB VSX Scalar Multiply-Subtract Type-A Double-Precision</td>
</tr>
<tr>
<td>\texttt{xsnmaddadp}</td>
<td>XT,XA,XB VSX Scalar Negative Multiply-Add Type-A Double-Precision</td>
</tr>
<tr>
<td>\texttt{xsnmsubadp}</td>
<td>XT,XA,XB VSX Scalar Negative Multiply-Subtract Type-A Double-Precision</td>
</tr>
<tr>
<td>\texttt{xvmaddasp}</td>
<td>XT,XA,XB VSX Vector Multiply-Add Type-A Single-Precision</td>
</tr>
<tr>
<td>\texttt{xvmsubasp}</td>
<td>XT,XA,XB VSX Vector Multiply-Subtract Type-A Single-Precision</td>
</tr>
<tr>
<td>\texttt{xvmnaddasp}</td>
<td>XT,XA,XB VSX Vector Negative Multiply-Add Type-A Single-Precision</td>
</tr>
<tr>
<td>\texttt{xvmsubasp}</td>
<td>XT,XA,XB VSX Vector Negative Multiply-Subtract Type-A Single-Precision</td>
</tr>
<tr>
<td>\texttt{xvmaddadp}</td>
<td>XT,XA,XB VSX Vector Multiply-Add Type-A Double-Precision</td>
</tr>
<tr>
<td>\texttt{xvmsubadp}</td>
<td>XT,XA,XB VSX Vector Multiply-Subtract Type-A Double-Precision</td>
</tr>
<tr>
<td>\texttt{xvmnaddadp}</td>
<td>XT,XA,XB VSX Vector Negative Multiply-Add Type-A Double-Precision</td>
</tr>
<tr>
<td>\texttt{xvmsubadp}</td>
<td>XT,XA,XB VSX Vector Negative Multiply-Subtract Type-A Double-Precision</td>
</tr>
</tbody>
</table>

Table 8. VSX Multiply-Add Arithmetic Instructions Providing Optimal Performance When Preceded by \texttt{xxlor}

Programming Note

Table 8 includes only the Type-A \textit{Multiply-Add} instructions because supporting only one of the two types (i.e. either Type-A or Type-M) is sufficient to preserve the contents of the destination operand of the permute or \textit{Multiply-Add} instruction. The \texttt{xxlor} instruction "preserves" the contents of the destination operand by copying it into another register, and the copy is then used as the destination operand of the \textit{Multiply-Add} instruction, which is overwritten upon execution.
2.2 Instruction Restart

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 6.6 of Book III.

### 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.

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.
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></td>
<td>(PPR32_{43:45})</td>
</tr>
<tr>
<td></td>
<td>001 very low</td>
</tr>
<tr>
<td></td>
<td>010 low</td>
</tr>
<tr>
<td></td>
<td>011 medium low</td>
</tr>
<tr>
<td></td>
<td>100 medium</td>
</tr>
<tr>
<td></td>
<td>101 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 4.3.8.) 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 4.3.7 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 \texttt{mfppr} and \texttt{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 \texttt{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 9. `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 9: 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 4.3.8 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.

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
13. Maximum transaction level

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.

<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Software Transient Enable (SWTE)</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Applies the transient attribute to software-defined streams.</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Hardware Transient Enable (HWTE)</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>HWTE is disabled.</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Store Transient Enable (STE)</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Applies the transient attribute to store streams.</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Load Transient Enable (LTE)</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Applies the transient attribute to load streams.</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Software Unit count Enable (SWUE)</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>SWUE is disabled.</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Hardware Unit count Enable (HWUE)</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>HWUE is disabled.</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Unit Count (UNITCNT)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Number of units in data stream.</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Load Stream Disable (LSD)</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>No effect.</td>
<td></td>
</tr>
</tbody>
</table>

Figure 2. Data Stream Control Register

This field indicates how quickly the prefetch depth should be reached for hardware-detected streams. Values and their meanings are as follows.

0 default
1 not urgent
2 least urgent
3 less urgent
4 medium
5 urgent
6 more urgent
7 most urgent
1 Disables hardware detection and initiation of load streams.

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 841, for information on streams and how software may specify them.

**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).

**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 mispredicts 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.
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 809, storage protection, reference and change recording, the storage access ordering described in Section 1.7.1, and Performance Monitor events (see Section 9.4.5 of Book III).

---

**Programming Note**

Accesses that are caused by or associated with *Cache Management* instructions that are "treated as a *Load*" or "treated as a *Store*" are not subject to the special ordering rules described for SAO storage. These accesses are always performed in accordance with the weakly consistent storage model.

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).

---

**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

---

**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.
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 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.

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.

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 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.
Stream ID (ID)

Stream ID to use for this data stream.

01010 The dcbit/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 dcbit/dcbtst instruction, or will probably no longer access such data streams.

The EA is interpreted as follows. If GO=1 and S≠0b00 the hint provided by the instruction is undefined; the remainder of this instruction description assumes that this combination is not used.

Bit(s) Description
0:31 Reserved
32 GO
0 No information is provided by the GO field.
1 For dcbit, 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. ("Nascent" and "completely described" are defined below.) For dcbtst, this field value holds no meaning and is treated as though it were zero.
33:34 Stop (S)
00 No information is provided by the S field.
01 Reserved
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.)

For dcbit, the program will probably no longer access the load and store data streams associated with all stream IDs. (All other fields of the EA are ignored.) For dcbtst, this field value holds no meaning, and is treated as though it were 0b00.

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.

0 default = DSCRDPFD
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).
The `dcbt/dcbtst` instruction provides a hint that describes certain attributes of a data stream.

The EA is interpreted as follows.

<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></td>
<td>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.</td>
</tr>
<tr>
<td>50</td>
<td>Reserved</td>
</tr>
<tr>
<td>51:55</td>
<td>Offset</td>
</tr>
<tr>
<td></td>
<td>The word-offset of the first element of the stream in its unit (i.e., the effective address of the first element of the stream is ((\text{EATRUNC}</td>
</tr>
<tr>
<td>56:59</td>
<td>Reserved</td>
</tr>
<tr>
<td>60:63</td>
<td>Stream ID (ID)</td>
</tr>
<tr>
<td></td>
<td>Stream ID to use for this data stream.</td>
</tr>
</tbody>
</table>

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.

01011 The `dcbt/dcbtst` instruction provides a hint that describes certain attributes of a data stream.

The EA is interpreted as follows.

<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></td>
<td>The displacement, in words, between the first byte of successive elements in the stream. The effective address of the (N)th 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.</td>
</tr>
<tr>
<td>50</td>
<td>Reserved</td>
</tr>
<tr>
<td>51:55</td>
<td>OFFSET</td>
</tr>
<tr>
<td></td>
<td>The word-offset of the first element of the stream in its unit (i.e., the effective address of the first element of the stream is ((\text{EATRUNC}</td>
</tr>
<tr>
<td>56:59</td>
<td>Reserved</td>
</tr>
<tr>
<td>60:63</td>
<td>Stream ID (ID)</td>
</tr>
<tr>
<td></td>
<td>Stream ID to use for this data stream.</td>
</tr>
</tbody>
</table>

Programming Note

A program should use a `dcbt/dcbtst` instruction with TH=0b01011 only when the stride is larger than 128 bytes. Otherwise, consecutive units will be accessed, so the additional stream information has no benefit.

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 \(\text{TH}=0b01011\), or (b) \(\text{TH}=0b01010\) 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=0b01010 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.

Interrupts (see Book III) cause all existing data streams to cease to exist. 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.

Programming Note
This Programming Note describes several aspects of using the data stream variants of the \texttt{dcbt} and \texttt{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 \texttt{dcbt} instruction. The corresponding specification for a data stream having other attributes requires two or three \texttt{dcbt/dcbtst} instructions to describe the stream and one additional \texttt{dcbt} instruction to start the stream. However, one \texttt{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 $2 \times n$ to $3 \times n$ \texttt{dcbt/dcbtst} instructions plus one \texttt{dcbt} instruction. (There is no need to execute a \texttt{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 \texttt{dcbt/dcbtst} instructions that describe data streams suffices.)

- If it is desired that the hint provided by a given \texttt{dcbt/dcbtst} instruction be provided in program order with respect to the hint provided by another \texttt{dcbt/dcbtst} instruction, the two instructions must be separated by an \texttt{eieio} instruction. For example, if a \texttt{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 \texttt{dcbt/dcbtst} instructions, and is intended \textit{not} to indicate that the program will probably soon access nascent data streams described (completely) by following \texttt{dcbt/dcbtst} instructions, an \texttt{eieio} instruction must separate the \texttt{dcbt} instruction with GO=1 from the preceding \texttt{dcbt/dcbtst} instructions, and another \texttt{eieio} instruction must separate that \texttt{dcbt} instruction from the following \texttt{dcbt/dcbtst} instructions.

- In practice, the second \texttt{eieio} described above can sometimes be omitted. For example, if the program consists of an outer loop that contains the \texttt{dcbt/dcbtst} instructions and an inner loop that contains the \textit{Load} or \textit{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 interrupts 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.

\textbf{TH=0b10000}

If TH=0b10000, the \texttt{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).

\textbf{TH=0b10001}

If TH=0b10001, the \texttt{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

dcbt  RA,RB,TH

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

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 the memory barrier created by a sync instruction.

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=0b01010 and TH=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

equivalent to:
dcbt for TH values of 0b00000 - 0b00111;
other TH values are invalid.
dcbtds RA,RB,TH
dcbt for TH values of 0b00000 or 0b01000 - 0b01111;
other TH values are invalid.
dcbtt RA,RB
dcbt for TH value of 0b10000

dcbna RA,RB
dcbt for TH value of 0b10001

Data Cache Block Touch for Store X-form

dcbtst  RA,RB,TH

Programming Notes

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.

Programming Note

See the Programming Notes at the beginning of this section.
Let the effective address (EA) be the sum (RA|0)+(RB).

The `dcbst` 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 `dcbst` 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 `dcbst` instruction (e.g., `dcbst` 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 `dcbst` 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≠0b01010 and TH≠0b01011, 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>dcbstct RA,RB,TH</th>
<th>Equivalent to:</th>
</tr>
</thead>
<tbody>
<tr>
<td>dcbst for TH values of 0b00000 or 0b00000 - 0b00111; other TH values are invalid.</td>
<td></td>
</tr>
</tbody>
</table>

| dcbststds RA,RB,TH | dcbst for TH values of 0b00000 or 0b01000 - 0b01111; other TH values are invalid. |
| dcbsttl RA,RB | dcbst for TH value of 0b10000. |

**Programming Note**

See the Programming Notes at the beginning of this section.

---

## Data Cache Block set to Zero

### X-form

<table>
<thead>
<tr>
<th>dcbbz</th>
<th>RA, RB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>31</td>
</tr>
<tr>
<td>31</td>
<td>30</td>
</tr>
<tr>
<td>30</td>
<td>29</td>
</tr>
<tr>
<td>29</td>
<td>28</td>
</tr>
<tr>
<td>28</td>
<td>27</td>
</tr>
<tr>
<td>27</td>
<td>26</td>
</tr>
<tr>
<td>26</td>
<td>25</td>
</tr>
<tr>
<td>25</td>
<td>24</td>
</tr>
<tr>
<td>24</td>
<td>23</td>
</tr>
<tr>
<td>23</td>
<td>22</td>
</tr>
<tr>
<td>22</td>
<td>21</td>
</tr>
<tr>
<td>21</td>
<td>20</td>
</tr>
<tr>
<td>20</td>
<td>19</td>
</tr>
<tr>
<td>19</td>
<td>18</td>
</tr>
<tr>
<td>18</td>
<td>17</td>
</tr>
<tr>
<td>17</td>
<td>16</td>
</tr>
<tr>
<td>16</td>
<td>15</td>
</tr>
<tr>
<td>15</td>
<td>14</td>
</tr>
<tr>
<td>14</td>
<td>13</td>
</tr>
<tr>
<td>13</td>
<td>12</td>
</tr>
<tr>
<td>12</td>
<td>11</td>
</tr>
<tr>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td>10</td>
<td>9</td>
</tr>
<tr>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TH</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01010</td>
<td></td>
</tr>
<tr>
<td>0b01011</td>
<td></td>
</tr>
</tbody>
</table>

if RA = 0 then b ← 0
else b ← (RA)
EA ← b + (RB)

If the block size is greater than 1, then:

\[ m = \log_2(n) \]

ea ← EA_{0x:m} || n0

MEN(ea, n) ← n0x00

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

`dcbbz` 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, `dcbbz` 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, `dcbbz` is likely to take significantly longer to execute than an equivalent sequence of Store instructions. For example, on some implementations dcbbz 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 5.9.1 of Book III for additional information about `dcbbz`. 

---

850 Power ISA™ II
Data Cache Block Store  X-form  

dcbst  RA,RB

<table>
<thead>
<tr>
<th>31</th>
<th>6</th>
<th>RA</th>
<th>RB</th>
<th>54</th>
<th>/</th>
</tr>
</thead>
</table>

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

Data Cache Block Flush  X-form  

dcbf  RA,RB,L

<table>
<thead>
<tr>
<th>31</th>
<th>L</th>
<th>RA</th>
<th>RB</th>
<th>86</th>
<th>/</th>
</tr>
</thead>
</table>

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.

For the L operand, the value 2 is reserved. The results of executing a dcbf instruction with L=2 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 911. The extended mnemonics are shown below.

Extended: Equivalent to:

dcbf RA,RB dcbf RA,RB,0

dcbfl RA,RB dcbf RA,RB,1

dcbflp RA,RB dcbf RA,RB,3

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 and "dcbflp" is used for L=3.

Programming Note

**dcbf** serves as both a basic and an extended mnemonic. The Assembler will recognize a **dcbf** mnemonic with three operands as the basic form, and a **dcbf** mnemonic with two operands as the extended form. In the extended form the L operand is omitted and assumed to be 0.

Programming Note

**dcbf** with L=1 can be used to provide a hint that a block in this processor’s data cache will not be reused soon.

**dcbf** with L=3 can be used to flush a block from the processor’s primary data cache but reduce the latency of a subsequent access. For example, the block may be evicted from the primary data cache but a copy retained in a lower level of the cache hierarchy.

Programs which manage coherence in software must use **dcbf** with L=0.

4.3.2.1 Obsolete Data Cache Instructions

The Data Stream Touch (dst), Data Stream Touch for Store (dstst), 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)).

4.3.3 “or” Instruction

**“or” Cache Control Hint**

**or** 26,26,26

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:

Extended: Equivalent to:

miso or 26,26,26

“miso” is short for “make it so.”
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:

```assembly
addi r1,r0,0x1234
sth r1,0x1000(r3)  # store data value 0x1234
lwsync             # order data store before flag store
addi r2,r0,0x0001
stb 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:

```assembly
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 flag load before # data load
lhz r1,0x1000(r3)  # load data value
lwsync             # order data load before # flag store
addi r2,r0,0x0000
stb r2,0x1002(r3)  # clear flag byte
or r26,r26,r26     # miso
```

---

**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.
4.4 Copy-Paste Facility

The Copy-Paste Facility provides a means to copy a block of data to an accelerator. It uses pairs of instructions, *copy* followed by *paste*, to define the data transfers. (See Section 1.7.2, “Storage Ordering of Copy/Paste-Initiated Data Transfers” for the memory model characteristics of these data transfers.) Authority to use an accelerator is established through a call to the hypervisor, the details of which are beyond the scope of the architecture. The format of the data block is accelerator-specific. The 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 0b0001x in CR0. Transient errors (a copy-paste sequence error, a memory management state change (tlbfl[)] during the transfer, or an implementation-specific transient problem) are indicated by a CR0 value of 0b0000x, 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, attempting to paste to normal memory, and 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**

A failure of a data transfer may be the result of a shortage of the resources required to complete the operation. 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.

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.

**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.
Copy

copy RA,RB

if RA = 0 then b ← 0
else b ← (RA)
EA ← b +(RB)
copy_buffer ← memory(EA,128)

Let the effective address (EA) be the sum (RA|0)+(RB).
The 128 bytes in storage 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 1.7.2, “Storage Ordering of Copy/Paste-Initiated Data Transfers”.

Special Registers Altered:
None

Paste

paste. RA,RB

if there was a copy-paste sequence error or a translation conflict
CR0←0b000||XERSO
else
  if RA = 0 then b ← 0
  else b ← (RA)
EA ← b +(RB)
post(memory(EA,128)) ← copy_buffer
wait for completion status
if there was a data transfer problem
CR0←0b000||XERSO
else
  CR0←0b001||XERSO
  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).

Post the contents of the copy buffer to be sent to the accelerator addressed by EA and wait for completion status on the data transfer. Set CR0 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>

Clear the state of the Copy-Paste Facility.

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.

If the associated copy specified an accelerator, if the paste specifies an accelerator that was not properly configured, or if the paste specifies normal storage, 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 1.7.2, “Storage Ordering of Copy/Paste-Initiated Data Transfers”.

Special Registers Altered:
CR0
**Copy-Paste Abort**  

This instruction is used to clear the state of the Copy-Paste Facility if a data transfer is in progress. Any pending errors in the Copy-Paste Facility are cleared, and the state is reset to prepare for a new copy.

**X-form**

```
  cpabort

31 6 11 16 21 838 31
```

This clears the state of the Copy-Paste Facility.

**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 both the weakly consistent and SAO storage models. 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 and AMOs 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 loads and fixed-point stores, and any reference elsewhere in the architecture to fixed-point loads or fixed-point stores apply to these instructions as well, except where explicitly stated otherwise 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 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 loads and stores is simply that the loads return a result to a GPR, while the stores 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.

---

**Engineering Note**

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>
</table>
| 00000         | RT, RT+1     | mem(EA,s)        | **Fetch and Add**<br>\[
\begin{align*}
  t & \leftarrow \text{mem}(EA, s) \\
  t2 & \leftarrow t + (RT+1) \\
  \text{mem}(EA,s) & \leftarrow t2 \\
  RT & \leftarrow t
\end{align*}
\] |
| 00001         | RT, RT+1     | mem(EA,s)        | **Fetch and XOR**<br>\[
\begin{align*}
  t & \leftarrow \text{mem}(EA, s) \\
  t2 & \leftarrow t \oplus (RT+1) \\
  \text{mem}(EA,s) & \leftarrow t2 \\
  RT & \leftarrow t
\end{align*}
\] |
| 00010         | RT, RT+1     | mem(EA,s)        | **Fetch and OR**<br>\[
\begin{align*}
  t & \leftarrow \text{mem}(EA, s) \\
  t2 & \leftarrow t \mid (RT+1) \\
  \text{mem}(EA,s) & \leftarrow t2 \\
  RT & \leftarrow t
\end{align*}
\] |
| 00011         | RT, RT+1     | mem(EA,s)        | **Fetch and AND**<br>\[
\begin{align*}
  t & \leftarrow \text{mem}(EA, s) \\
  t2 & \leftarrow t \& (RT+1) \\
  \text{mem}(EA,s) & \leftarrow t2 \\
  RT & \leftarrow t
\end{align*}
\] |
| 00100         | RT, RT+1     | mem(EA,s)        | **Fetch and Maximum Unsigned**<br>\[
\begin{align*}
  t & \leftarrow \text{mem}(EA, s) \\
  \text{if } (RT+1) > u t & \text{ then } \text{mem}(EA,s) \leftarrow (RT+1) \\
  RT & \leftarrow t
\end{align*}
\] |
| 00101         | RT, RT+1     | mem(EA,s)        | **Fetch and Maximum Signed**<br>\[
\begin{align*}
  t & \leftarrow \text{mem}(EA, s) \\
  \text{if } (RT+1) > t & \text{ then } \text{mem}(EA,s) \leftarrow (RT+1) \\
  RT & \leftarrow t
\end{align*}
\] |
| 00110         | RT, RT+1     | mem(EA,s)        | **Fetch and Minimum Unsigned**<br>\[
\begin{align*}
  t & \leftarrow \text{mem}(EA, s) \\
  \text{if } (RT+1) < u t & \text{ then } \text{mem}(EA,s) \leftarrow (RT+1) \\
  RT & \leftarrow t
\end{align*}
\] |
| 00111         | RT, RT+1     | mem(EA,s)        | **Fetch and Minimum Signed**<br>\[
\begin{align*}
  t & \leftarrow \text{mem}(EA, s) \\
  \text{if } (RT+1) < t & \text{ then } \text{mem}(EA,s) \leftarrow (RT+1) \\
  RT & \leftarrow t
\end{align*}
\] |
| 01000         | RT, RT+1     | mem(EA,s)        | **Swap**<br>\[
\begin{align*}
  t & \leftarrow \text{mem}(EA, s) \\
  \text{mem}(EA,s) & \leftarrow (RT+1) \\
  RT & \leftarrow t
\end{align*}
\] |
| 10000         | RT, RT1, RT+2| mem(EA,s)        | **Compare and Swap Not Equal**<br>\[
\begin{align*}
  t & \leftarrow \text{mem}(EA, s) \\
  \text{if } t != (RT+1) & \text{ then } \text{mem}(EA,s) \leftarrow (RT+2) \\
  RT & \leftarrow t
\end{align*}
\] |
| 11000         | RT           | mem(EA,s)        | **Fetch and Increment Bounded**<br>\[
\begin{align*}
  t & \leftarrow \text{mem}(EA, s) \\
  t2 & \leftarrow \text{mem}(EA+s, s) \\
  \text{if } t != t2 & \text{ then } \\
  \text{mem}(EA,s) & \leftarrow t+1 \\
  RT & \leftarrow t \\
  \text{else } RT & \leftarrow 1 << (s*8-1)
\end{align*}
\] |
Figure 3. Load Atomic function codes

<table>
<thead>
<tr>
<th>Function Code</th>
<th>RT</th>
<th>mem(EA, s)</th>
<th>mem(EA+s, s)</th>
<th>Fetch and Increment Equal</th>
</tr>
</thead>
<tbody>
<tr>
<td>11001</td>
<td>RT</td>
<td>mem(EA, s)</td>
<td>mem(EA+s, s)</td>
<td>t ← mem(EA, s)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>t2 ← mem(EA+s, s)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>if t = t2 then</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mem(EA, s) ← t+1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>RT ← t</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>else RT ← 1 &lt;&lt; (s*8-1)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Function Code</th>
<th>RT</th>
<th>mem(EA-s, s)</th>
<th>mem(EA, s)</th>
<th>Fetch and Decrement Bounded</th>
</tr>
</thead>
<tbody>
<tr>
<td>11100</td>
<td>RT</td>
<td>mem(EA-s, s)</td>
<td>mem(EA, s)</td>
<td>t ← mem(EA, s)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>t2 ← mem(EA+s, s)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>if t != t2 then</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mem(EA, s) ← t-1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>RT ← t</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>else RT ← 1 &lt;&lt; (s*8-1)</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.
Load Word Atomic X-form

lwat RT, RA, FC

<table>
<thead>
<tr>
<th>0</th>
<th>31</th>
<th>RT</th>
<th>RA</th>
<th>FC</th>
<th>582</th>
</tr>
</thead>
</table>

if RA=0 then EA ← 0
else EA ← (RA)

(RT|32:63, mem(EA, 4)) ← lat(mem(EA-4, 12), RT|32:63, RT+1|32:63, FC)

RT|0:31 ← 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 lwat specifying an invalid function code will cause the system data storage error handler to be invoked.

The portion of mem(EA-4, 12) accessed by the instruction must be contained within an aligned 32-byte block of storage. If it is not, the system alignment error handler will be invoked.

Special Registers Altered:
None

Load Doubleword Atomic X-form

ldat RT, RA, FC

<table>
<thead>
<tr>
<th>0</th>
<th>31</th>
<th>RT</th>
<th>RA</th>
<th>FC</th>
<th>614</th>
</tr>
</thead>
</table>

if RA=0 then EA ← 0
else EA ← (RA)

(RT, mem(EA, 8)) ← lat(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 ldat specifying an invalid function code will cause the system data storage error handler to be invoked.

The portion of mem(EA-8, 24) accessed by the instruction must be contained within an aligned 32-byte block of storage. If it is not, the system alignment error handler will be 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}(\text{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>\text{mem}(\text{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}(\text{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>\text{mem}(\text{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}(\text{EA}, s) )</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>( t2 \leftarrow t</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>\text{mem}(\text{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}(\text{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>\text{mem}(\text{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}(\text{EA}, s) )</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>\text{if} (RS) &gt;^u t \text{ then } \text{mem}(\text{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}(\text{EA}, s) )</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>\text{if} (RS) &gt; t \text{ then } \text{mem}(\text{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}(\text{EA}, s) )</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>\text{if} (RS) &lt;^u t \text{ then } \text{mem}(\text{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}(\text{EA}, s) )</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>\text{if} (RS) &lt; t \text{ then } \text{mem}(\text{EA},s) \leftarrow (RS)</td>
</tr>
<tr>
<td>11000</td>
<td>RS</td>
<td>mem(EA,s)</td>
<td>Store Twin</td>
</tr>
<tr>
<td></td>
<td></td>
<td>mem(EA+s, s)</td>
<td>( t \leftarrow \text{mem}(\text{EA}, s) )</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>( t2 \leftarrow \text{mem}(\text{EA}+s, s) )</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>\text{if} t = t2 \text{ then }</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>\text{mem}(\text{EA},s) \leftarrow (RS)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>\text{mem}(\text{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

| 0 | 31 | RS | RA | FC | 710 |

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.

The portion of mem(EA,8) accessed by the instruction must be contained within an aligned 32-byte block of storage. If it is not, the system alignment error handler will be invoked.

**Special Registers Altered:** None

---

**Store Doubleword Atomic X-form**

stdat RS,RA,FC

| 0 | 31 | RS | RA | FC | 742 |

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.

The portion of mem(EA,16) accessed by the instruction must be contained within an aligned 32-byte block of storage. If it is not, the system alignment error handler will be 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

\( \text{isync} \)

Executing an \( \text{isync} \) instruction ensures that all instructions preceding the \( \text{isync} \) instruction have completed before the \( \text{isync} \) instruction completes, and that no subsequent instructions are initiated until after the \( \text{isync} \) instruction completes. It also ensures that all instruction cache block invalidations caused by icbi instructions preceding the \( \text{isync} \) instruction have been performed with respect to the processor executing the \( \text{isync} \) instruction, and then causes any prefetched instructions to be discarded.

Except as described in the preceding sentence, the \( \text{isync} \) instruction may complete before storage accesses associated with instructions preceding the \( \text{isync} \) 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 1.7.4, “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.
Load Byte And Reserve Indexed X-form

<table>
<thead>
<tr>
<th>Symbol</th>
<th>RT</th>
<th>RA</th>
<th>RB</th>
<th>EH</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>31</td>
<td>5</td>
<td>11</td>
<td>16</td>
</tr>
<tr>
<td>1</td>
<td>21</td>
<td>31</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + (RB)
RESERVE ← 1
RESERVE_LENGTH ← 1
RESERVE_ADDR ← real_addr(EA)
RT ← 560 || MEM(EA, 1)

Let the effective address (EA) be the sum (RA|0)+(RB). The byte in storage addressed by EA is loaded into RT56:63. RT0:55 are set to 0.

This instruction creates a reservation for use by a stbcx instruction. A real address computed from the EA as described in Section 1.7.4.1 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.

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.

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 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

Either value of the EH field is appropriate for a Load and Reserve instruction that is intended to establish a reservation for a subsequent waitrsv and not a subsequent Store Conditional instruction.

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 Halfword And Reserve Indexed X-form

\[ \text{lharx } \text{RT,RA,RB,EH} \]

<table>
<thead>
<tr>
<th>31</th>
<th>RT</th>
<th>RA</th>
<th>RB</th>
<th>116</th>
<th>EH</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

if RA = 0 then b ← 0
else           b ← (RA)
EA ← b +(RB)
RESERVE ← 1
RESERVE_LENGTH ← 2
RESERVE_ADDR ← real_addr(EA)
RT ← 4K0 || 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 instruction. A real address computed from the EA as described in Section 1.7.4.1 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 ---
\text{lharx} serves as both a basic and an extended mnemonic. The Assembler will recognize a \text{lharx} mnemonic with four operands as the basic form, and a \text{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

\[ \text{lwarx } \text{RT,RA,RB,EH} \]

<table>
<thead>
<tr>
<th>31</th>
<th>RT</th>
<th>RA</th>
<th>RB</th>
<th>20</th>
<th>EH</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

if RA = 0 then b ← 0
else           b ← (RA)
EA ← b +(RB)
RESERVE ← 1
RESERVE_LENGTH ← 4
RESERVE_ADDR ← real_addr(EA)
RT ← 12K | 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 instruction. A real address computed from the EA as described in Section 1.7.4.1 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 ---
\text{lwarx} serves as both a basic and an extended mnemonic. The Assembler will recognize a \text{lwarx} mnemonic with four operands as the basic form, and a \text{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>23</th>
<th>11</th>
<th>6</th>
<th>21</th>
<th>16</th>
<th>694</th>
<th>1</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>RS</td>
<td>RA</td>
<td>RB</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 = 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 || u1 || u2 || XER_SO
else
   CR0 ← 0b00 || store_performed || XER_SO
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).

CR0LT GT EQ SO = 0b00 || n || XER_SO

The reservation is cleared.

Special Registers Altered:
CR0
Store Halfword Conditional Indexed X-form

\texttt{sthcx. RS,RA,RB}

\begin{verbatim}
if RA = 0 then b \leftarrow 0
else b \leftarrow (RA)
EA \leftarrow b + (RB)
if RESERVE then
  if RESERVE_LENGTH = 2 &
    RESERVE_ADDR = real_addr(EA) then
      MEM(EA, 2) \leftarrow (RS)_{48:63}
      undefined_case \leftarrow 0
    store_performed \leftarrow 1
  else
    z \leftarrow \text{smallest real page size supported by implementation}
    if RESERVE_ADDR \div z = real_addr(EA) \div z then
      undefined_case \leftarrow 1
    else
      undefined_case \leftarrow 0
      store_performed \leftarrow 0
else
  undefined_case \leftarrow 0
  store_performed \leftarrow 0
if undefined_case then
  u1 \leftarrow \text{undefined 1-bit value}
  if u1 then
    MEM(EA, 2) \leftarrow (RS)_{48:63}
  u2 \leftarrow \text{undefined 1-bit value}
  CR0 \leftarrow 0b00 | | u2 | | XER_{SO}
else
  CR0 \leftarrow 0b00 | | store_performed | | XER_{SO}
RESERVE \leftarrow 0
\end{verbatim}

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 \texttt{sthcx.} is the same as the real storage location specified by the \texttt{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 \texttt{sthcx.} is not the same as the real storage location specified by the \texttt{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 \texttt{sthcx.} is in the same aligned z-byte block of real storage as the real storage location specified by the \texttt{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).

\[ \text{CR0}_{LT\,GT\,EQ\,SO} = 0b00 | | n | | XER_{SO} \]

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:

\texttt{CR0}
Store Word Conditional Indexed X-form

The store is performed, the value of \( n \) is undefined (and need not reflect whether the store was performed).

\[
\text{CR0}_{\text{LT GT EQ SO}} = 0b00 \parallel n \parallel \text{XER}_{\text{SO}}
\]

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

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 \textit{stwcx} is the same as the real storage location specified by the \textit{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 \textit{stwcx} is not the same as the real storage location specified by the \textit{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 \textit{stwcx} is in the same aligned \( z \)-byte block of real storage as the real storage location specified by the \textit{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
4.6.2.1 64-Bit Load and Reserve and Store Conditional Instructions

**Load Doubleword And Reserve Indexed X-form**

```
ldarx RT, RA, RB, EH
```

1. If RA = 0 then b ← 0
2. Else b ← (RA)
3. EA ← b + (RB)
4. RESERVE ← 1
5. RESERVE_LENGTH ← 8
6. RESERVE_ADDR ← real_addr(EA)
7. 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` instruction. A real address computed from the EA as described in Section 1.7.4.1 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**

`ldarx` serves as both a basic and an extended mnemonic. The Assembler will recognize a `ldarx` 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.

**Store Doubleword Conditional Indexed X-form**

```
stdcx. RS, RA, RB
```

1. If RA = 0 then b ← 0
2. Else b ← (RA)
3. EA ← b + (RB)
4. If RESERVE then
   1. If RESERVE_LENGTH = 8 & RESERVE_ADDR = real_addr(EA) then
      MEM(EA, 8) ← (RS)
      undefined_case ← 0
      store_performed ← 1
   2. 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
   3. Else
      undefined_case ← 0
      store_performed ← 0
5. If undefined_case then
   1. u1 ← undefined 1-bit value
   2. If u1 then
      MEM(EA, 8) ← (RS)
      u2 ← undefined 1-bit value
      CR0 ← 0b00 || u2 || XER_S0
   2. Else
      CR0 ← 0b00 || store_performed || XER_S0
      RESERVE ← 0

Let the effective address (EA) be the sum (RA|0)+(RB).

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 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, the following applies.

Let z denote the smallest real page size supported by the implementation. If the real storage location specified by the `stdcx`, is in the same aligned z-byte block of real storage as the real storage location specified by the `ldarx` instruction that established the reservation, it is undefined whether (RS) is stored into the doubleword 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).

\[
\text{CR0}_{\text{LT GT EQ SO}} = 0b00 \ || \ n \ || \ \text{XER}_{\text{SO}}
\]

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

---

**Power ISA™ II**
4.6.2.2 128-bit Load and Reserve Store Conditional Instructions

For \textit{\texttt{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 \(RB \neq RTp+1\).

**Load Quadword And Reserve Indexed X-form**

\texttt{lqarx RTp,RA,RB,EH}

| 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 |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0   | 6  | RA | 16 | RB | 276| EH |

- 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{\texttt{stqcx}} instruction. A real address computed from the EA as described in Section 1.7.4.1 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{\texttt{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.

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.

For \textit{\texttt{stqcx}}, 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.

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 \(RTp=RA=0\).)

**Special Registers Altered:**

- None

---

**Programming Note**

\textit{\texttt{lqarx}} serves as both a basic and an extended mnemonic. The Assembler will recognize a \textit{\texttt{lqarx}} mnemonic with four operands as the basic form, and a \textit{\texttt{lqarx}} mnemonic with three operands as the extended form. In the extended form the EH operand is omitted and assumed to be 0.
**Store Quadword Conditional Indexed X-form**

\[
\text{stqcx.}\ RSp,\ RA,\ RB
\]

<table>
<thead>
<tr>
<th></th>
<th>P</th>
<th>RSp</th>
<th>RA</th>
<th>RB</th>
<th>182</th>
<th>21</th>
<th>16</th>
<th>11</th>
<th>10</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></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>
</tr>
</tbody>
</table>

if RA = 0 then \(b \leftarrow 0\)
else \(b \leftarrow (\text{RA})\)

**EA** \(\leftarrow b + (\text{RB})\)

if RESERVE then

if \(\text{RESERVE\_LENGTH} = 16\) &
\(\text{RESERVE\_ADDR} = \text{real\_addr}(\text{EA})\) then
\[
\text{MEM}(\text{EA}, 16) \leftarrow (\text{RSp})
\]
undefined_case \(\leftarrow 0\)
store_performed \(\leftarrow 1\)
else
\[
z \leftarrow \text{smallest real page size supported by implementation}
\]
if \(\text{RESERVE\_ADDR} + z = \text{real\_addr}(\text{EA})\) then
undefined_case \(\leftarrow 1\)
else
undefined_case \(\leftarrow 0\)
store_performed \(\leftarrow 0\)
else
undefined_case \(\leftarrow 0\)
store_performed \(\leftarrow 0\)

if undefined_case then
\[
\text{u1} \leftarrow \text{undefined 1-bit value}
\]
if \(\text{u1}\) then
\[
\text{MEM}(\text{EA}, 16) \leftarrow (\text{RSp})
\]
\(\text{u2} \leftarrow \text{undefined 1-bit value}\)
\[
\text{CR0} \leftarrow 0b00 \parallel \text{u2} \parallel \text{XERSO}
\]
else
\[
\text{CR0} \leftarrow 0b00 \parallel \text{store\_performed} \parallel \text{XERSO}
\]
RESERVE \(\leftarrow 0\)

Let the effective address (EA) be the sum \((\text{RA}(0)+(\text{RB}))\).

If a reservation exists, the length associated with the reservation is 16 bytes, and the real storage location specified by the **stqcx.** is the same as the real storage location specified by the **lqarx** instruction that established the reservation, \((\text{RSp})\) is stored into the quadword in storage addressed by \(\text{EA}\).

If a reservation exists, and either the length associated with the reservation is not 16 bytes or the real storage location specified by the **stqcx.** is not the same as the real storage location specified by the **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 **stqcx.** is in the same aligned \(z\)-byte block of real storage as the real storage location specified by the **lqarx** instruction that established the reservation, it is undefined whether \((\text{RSp})\) is stored into the quadword in storage addressed by \(\text{EA}\). 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).

\[
\text{CR0}_{\text{LT GT EQ SO}} = 0b00 \parallel n \parallel \text{XERSO}
\]
The reservation is cleared.

\(\text{EA}\) 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 \(\text{RSp}\) 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. See Section 1.8, “Transactions” for a description of how the Memory Barrier instructions interact with transactions. Additional information about these instructions and about related aspects of storage management can be found in Book III.

The Memory Barrier instructions can be used to control the order in which storage accesses and data transfers are performed. See Section 1.8, “Transactions” for a description of how the Memory Barrier instructions interact with transactions. 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>sync</th>
<th>L</th>
<th>X-form</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>//</td>
<td>L</td>
</tr>
<tr>
<td>6</td>
<td>9</td>
<td>11</td>
</tr>
<tr>
<td>21</td>
<td>598</td>
<td>/</td>
</tr>
</tbody>
</table>

switch(L)
- case(0): hwsync
- case(1): lwsync
- case(2): ptesync

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 field as follows.

- **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. The applicable pairs are all pairs \(a_i, b_j\) of 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 storage accesses except those in which \(a_i\) is an 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 5.9.2 of Book III, as are additional properties of the sync instruction with L=2.

The ordering done by the memory barrier is cumulative (regardless of L value).

If L=0 (or L=2), the sync instruction has the following additional properties.

- Executing the sync instruction ensures that all instructions preceding the sync instruction have completed before the sync instruction completes, and that no subsequent instructions are initiated until after the sync instruction completes.

- The 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 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 icbi instruction executed by the processor executing the sync instruction (P1) may not have been performed with respect to P1 (see the description of the icbi instruction on page 840).

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.

**Programming Note**

Section 1.9 contains a detailed description of how to modify instructions such that a well-defined result is obtained.

The value L=3 is reserved.

The sync instruction may complete before storage accesses associated with instructions preceding the sync instruction have been performed.

**Special Registers Altered:**

None
Extended Mnemonics:

Extended mnemonics for *Synchronize*:

<table>
<thead>
<tr>
<th>Extended</th>
<th>Equivalent to</th>
</tr>
</thead>
<tbody>
<tr>
<td>sync</td>
<td>sync 0</td>
</tr>
<tr>
<td>lwsync</td>
<td>sync 1</td>
</tr>
<tr>
<td>ptesync</td>
<td>sync 2</td>
</tr>
</tbody>
</table>

Except in the *sync* instruction description in this section, references to "*sync*" in Books I-III imply L=0 unless otherwise stated or obvious from context; the appropriate extended mnemonics are used when other L values are intended.

---

**Programming Note**

*sync* serves as both a basic and an extended mnemonic. Assemblers will recognize a *sync* mnemonic with one operand as the basic form, and a *sync* mnemonic with no operand as the extended form. In the extended form the L operand is omitted and assumed to be 0.

---

**Programming Note**

The *sync* instruction can be used to ensure that all stores into a data structure, caused by *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 915.

The memory barrier created by a *sync* instruction with L=1 does not order implicit storage accesses or instruction fetches. The memory barrier created by a *sync* instruction with L=0 (or L=2) orders implicit storage accesses and instruction fetches associated with instructions preceding the *sync* instruction but not those associated with instructions following the *sync* instruction.

In order to obtain the best performance across the widest range of implementations, the programmer should use the *sync* instruction with L=1, or the *eieio* instruction, if any of these is sufficient for his needs; otherwise he should use *sync* with L=0. *sync* with L=2 should not be used by application programs.

---

**Programming Note**

The functions provided by *sync* with L=1 are a strict subset of those provided by *sync* with L=0. (The functions provided by *sync* with L=2 are a strict superset of those provided by *sync* with L=0; see Book III.)
Enforce In-order Execution of I/O X-form

`eieio`

The `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 `Load`, `Store`, and `dcbz` instructions executed by the processor executing the `eieio` instruction. These storage accesses are divided into the two sets listed below. The storage access caused by a `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 `dcbt` and `dcbtst` instructions (i.e., the providing of hints) are ordered by `eieio` as a third set of operations, the operations caused by `tlbie` and `tlbsync` instructions (see Book III) are ordered by `eieio` as a fourth set of operations, and the operations caused by `slbieg` or `slbiag` and `slbsync` instructions (see Book III) are ordered by `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 `eieio`'s memory barrier for the second set is cumulative; the ordering done by `eieio`'s memory barrier for the other four sets is not cumulative.

The `eieio` instruction may complete before storage accesses associated with instructions preceding the `eieio` instruction have been performed. The `eieio` instruction may complete before operations caused by `dcbt` and `dcbtst` instructions preceding the `eieio` instruction have been performed.

**Special Registers Altered:** None

### 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), and task completion by accelerators in the platform (referred to as “platform notify”).
The `wait` instruction causes instruction fetching and execution to be suspended. Instruction fetching and execution are resumed when the events specified by the WC field occur.

The values of the WC field are as follows.

- **0b00**: Resume instruction fetching and execution when an exception, an event-based branch exception, or a platform notify occurs.
- **0b01:11**: Reserved.
- **The exception, EBB exception, or platform notify causes the `wait` instruction to complete and instruction fetching to resume.**

When the `wait` instruction completes, processing is resumed either at the instruction following the `wait` (if interrupts and/or event-based branches are disabled) or in the corresponding interrupt or event-based branch handler (if interrupts and/or event-based branches are enabled). If an interrupt or event-based branch causes resumption of instruction execution, the interrupt or event-based branch handler will return to the instruction after the `wait`.

**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</td>
</tr>
</tbody>
</table>

**Programming Note**

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 6.4 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 a platform notify occurs.

Also, since exceptions corresponding to interrupts can cause `wait` to resume at any time without any EBB exception or platform notify having occurred, 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
```

**Programming Note**

Applications that execute `wait` in order to suspend processing until an external event-based branch exception occurs (see Section 7.2) 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.

**Programming Note**

The values in LPIDR, PIDR, and TIDR uniquely identify a thread that has initiated processing on an accelerator. Platforms may use these resources to track the locations of threads in the system. This service enables an accelerator to cause its initiating thread to resume processing when its results are available.
Chapter 5. Transactional Memory Facility

5.1 Transactional Memory Facility Overview

This chapter describes the registers and instructions that make up the transactional memory (TM) facility. Transactional memory is a shared-memory synchronization construct allowing an application to perform a sequence of storage accesses that appear to occur atomically with respect to other threads.

A set of instructions, special-purpose registers, and state bits in the MSR (see Book III) are used to control a transactional facility that is associated with each hardware thread. A `tbegin` instruction is used to initiate transactional execution, and a `tend` instruction is used to terminate transactional execution. Loads and stores that occur between the `tbegin` and `tend` instructions appear to occur atomically. An implementation may prematurely terminate transactional execution for a variety of reasons, rolling back all transactional storage updates that have been made by the thread since the `tbegin` was executed, and rolling back the contents of a subset of the thread's Book I registers to their contents before the `tbegin` was executed. In the event of such premature termination, control is transferred to a software failure handler associated with the transaction, which may then retry the transaction or choose an alternate path depending on the cause of transaction failure. A transaction can be explicitly aborted via a set of `conditional abort` instructions and an `unconditional abort` instruction, `tabort`. A `tsr` instruction is used to suspend or resume transactional execution, while allowing the transaction to remain active.

---

**Programming Note**

A `tbegin` should always be followed immediately by a `beq` as the first instruction of the failure handler, that branches to the main body of the failure handler. The failure handler should always either retry the transaction or use non-transactional code to perform the same operation. (The number of retries should be limited to avoid the possibility of an infinite loop. The limit could be based on the perceived permanence / transience of the failure.) A failure handler policy which includes trying a different transaction before returning to the one that failed may fail to make forward progress.

---

In code that may be executed transactionally, conditional branches should hint in favor of successful transactional execution where such a distinction exists. For example, the branch immediately following `tbegin` should hint that the branch is very likely not to be taken. As another example, consider a method of coding a failure handler that executes the body of a transaction non-transactionally by branching past the TM control instructions (e.g. `tsuspend`). Branches that bypass the TM control instructions should also hint that the branch is very likely not to be taken. These predictions will improve the efficiency of transactional execution, and may also help prevent the addition of spurious accesses to the transactional footprint.
Transactions performed using this facility are "strongly atomic", meaning that they appear atomic with respect to both transactional and non-transactional accesses performed by other threads. Transactions are isolated from reads and writes performed by other threads; i.e., transactional reads and writes will not appear to be interleaved with the reads and writes of other threads.

Nesting of transactions is supported using a form of nesting called "flattened nesting," in which transactions that are initiated during transactional execution are subsumed by the pre-existing transaction. Consequently, the effects of a nested transaction do not become visible until the outer transaction commits, and if a nested transaction fails, the entire set of transactions (outer as well as nested) is rolled back, and control is transferred to the outer transaction's failure handler. The memory barriers created by \texttt{tbegin} and \texttt{tend}, and the integrated cumulative memory barrier that are described in Section 1.8, "Transactions" are only created for outer transactions and not for any transactions nested within them.

References to \texttt{Store} instructions, and stores, include \texttt{dcbz} and the storage accesses that it causes.

\section*{Rollback-Only Transactions}

Rollback-Only Transactions (ROTs) differ from normal transactions in that they are speculative but not atomic. They are initiated by a unique variant of \texttt{tbegin}. They may be nested with other ROTs or with normal transactions. When a normal transaction is nested within a ROT, the behavior from the normal \texttt{tbegin}, until the end of the outer transaction is characteristic of a normal transaction. Although subject to failure from storage conflicts, the typical cause of ROT failure is via a \texttt{Tabort} variant that is executed after the program detects an error in its (software) speculation. Except where specifically differentiated or where differences follow from specific differentiation, the following description applies to ROTs as well as normal transactions.

\section*{5.1.1 Definitions}

\textbf{Commit}: A transaction is said to commit when it successfully completes execution. When a transaction is committed, its transactional accesses become irrevocable, and are made visible to other threads. A transaction completes by either committing or failing.

\textbf{Checkpointed registers}: The set of registers that are saved to the "checkpoint area" when a transaction is initiated, and restored upon transaction failure, is a subset of the architected register state, consisting of the General Purpose Registers, Floating-Point Registers, Vector Registers, Vector-Scalar Registers, and the following special registers and fields: CR fields other than CR0, XER, LR, CTR, FPSCR, AMR, PPR, VRSAVE, VSCR, DSCR, and TAR. The checkpointed registers include all problem state writable registers with the exception of CR0, EBBHR, EBBRR, BESCR, the Performance Monitor registers, and the Transactional Memory registers. With the exception of updates of CR0, the GSR, and the Transactional Memory registers, explicit updates of registers that are not included in the set of checkpointed registers are disallowed in Transactional state (i.e., will cause the transaction to fail), but are permitted in Suspended state. Suspended state modifications of these registers will not be rolled back in the event of transaction failure. Modifications of Transactional Memory registers are permitted in Non-transactional state, and modifications of the TFHAR are also permitted in Suspended state. Other attempts to modify Transactional Memory registers will cause a TM Bad Thing type Program interrupt.)

\begin{figure}[h]
\centering
\begin{tabular}{|c|c|}
\hline
\textbf{Programming Note} & \textbf{Programming Note} \\
\hline
The architecture does not include a “fairness guarantee” or a “forward progress” guarantee for transactions. If two processors repeatedly conflict with one another in an attempt to complete a transaction, one of the two may always succeed while the other may always fail. If two processors repeatedly conflict with one another in an attempt to complete a transaction, both may always fail, depending on the details of the transaction. This is different from the behavior of a typical locking routine, in which one or the other of the competitors will generally get the lock. & The GSR is not checkpointed because it has no contents; see Chapter 11 of Book III. \\
\hline
\end{tabular}
\caption{Example of a table.}
\end{figure}
Transactional accesses: Data accesses that are caused by an instruction that is executed when the thread is in the Transactional state (see Section 5.2) are said to be “transactional,” or to have been “performed transactionally.” The set of accesses caused by a committed normal transaction is performed as if it were a single atomic access. That is, it is always performed in its entirety with no visible fragmentation. The sets performed by normal transactions 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. Until a transaction commits, its set of transactional accesses is provisional, and will be discarded should the transaction fail. The set of transactional accesses is also referred to as the “transactional footprint.”

Non-transactional accesses: Storage accesses performed in the existing Power storage model are said to be “non-transactional.” In contrast to transactional storage accesses, there is no provision of atomicity across multiple non-transactional accesses. Non-transactional storage updates are not discarded in the event of a transaction failure.

Outer transaction: A transaction that is initiated from the Non-transactional state is said to be an outer transaction. A `tbegin` instruction that initiates an outer transaction is sometimes referred to as an “outer `begin`.” Similarly, a `tend` instruction with A=0 that ends an outer transaction is sometimes referred to as an “outer `end`.”

Nested Transaction: A transaction that is initiated while already executing a transaction is said to be “nested” within the pre-existing transaction. The set of active nested transactions forms a stack growing from the outer transaction. A `tend` with A=0 will remove the most recently nested transaction from the stack.

Failure: A transaction failure is an exceptional condition causing the transactional footprint to be discarded, the checkpointed registers to be reverted to their pre-transactional values, and the failure handler to get control.

Failure handler: A failure handler is a software component responsible for handling transaction failure. On transaction failure, hardware redirects control to the failure handler associated with the outer transaction.

Conflict: A transactional storage access is said to conflict with another transactional or non-transactional storage access if the two accesses overlap—i.e. if there is at least one byte that is referenced by both accesses—and at least one of the accesses is a store. If two transactions make conflicting accesses, at least one of them will fail. If a transaction fails as a result of a conflict with a store, the store may have been executed by another processor or may have been executed in Suspended state by the processor with the failing transaction. For a ROT, no conflict is caused if the ROT performs a load and another program performs a non-transactional store to the same storage location. The granularity at which conflict between storage accesses is detected is implementation-dependent, and may vary between accesses, but is never larger than a cache block.

A transactional storage access is said to conflict with a `tlbie` or `slibie` if the storage location being accessed is in the page or segment the translation for which is being invalidated by the `tlbie` or `slibie`. For a ROT, no conflict is caused if the access is a load.

A Suspended state cache control instruction is said to cause a conflict if it would cause the destruction of a transactional update or if it would make a transactional update visible to another thread.
explicitly entered with the execution of a Suspended Section 5.3.3.

table is transferred to the failure handler as described in failure is recorded as defined in Section 5.3.2, and con-
non-transactionally. In the event of transaction failure, transactions executed in the Transactional state are performed
Storage accesses (data accesses) caused by instructions executed in the Transactional state. Storage accesses and accesses to
interaction as well as any nested transactions, should they exist.
Non-transactional: The default, initial state of execution; no transaction is executing. The transactional facility is available for the initiation of a new transaction.

Transactional: This state is initiated by the execution of a tsr instruction in the Non-transactional state. Storage accesses (data accesses) caused by instructions executed in the Transactional state are performed transactionally. Other storage accesses associated with instructions executed in the Transactional state (instruction fetches, implicit accesses) are performed non-transactionally. In the event of transaction failure, failure is recorded as defined in Section 5.3.2, and control is transferred to the failure handler as described in Section 5.3.3.

Suspended: The Suspended execution state is explicitly entered with the execution of a tsrcp instruction during a transaction, the execution of a tsrcp instruction from Non-transactional state, or as a side-effect of an interrupt while in the Transactional state. Storage accesses and accesses to SPRs that are not part of the checkpointed registers are performed non-transactionally; they will be performed independently of the outcome of the transaction. The initiation of a new transaction is prevented in this state. In the event of transaction failure, failure recording is performed as described in Section 5.3.2, but failure handling is usually deferred until transactional execution is resumed (see Section 5.3.3 for details).

Until failure occurs, Load instructions that access storage locations that were transactionally written by the same thread will return the transactionally written data. After failure is detected, but before failure handling is performed, such loads may return either the transactionally written data, or the current non-transactional contents of the accessed location. The tcheck instruction can be used to determine whether any previous such loads may have returned non-transactional contents.

Suspended state Store instructions that access storage locations that have been accessed transactionally (due to load or store) by the same thread cause the transaction to fail.

--- Programming Note ---

Warning: In descriptions of the transactional memory facility that precede V. 2.07B, the granularity at which conflict between storage accesses is detected was specified to be the cache block. Programs that were based on these early descriptions and depend on this granularity may need to be revised so as not to depend on it.

A future version of the architecture may define "transaction conflict granule", as the aligned unit of storage having the property that the granularity at which conflict between storage accesses is detected is never larger than the transaction conflict granule. The size of the transaction conflict granule would be implementation-dependent and would be added to the list of parameters useful to application programs in Section 4.1 and the last sentence of the first paragraph of the definition of "conflict" would use "transaction conflict granule" instead of "cache block".

--- Programming Note ---

The intent of the Suspended execution state is to temporarily escape from transactional handling when transactional semantics are undesirable. Examples of such cases include storage updates that should be retained in the event of transactional failure, which is useful for debugging, interthread communication, the access of Caching Inhibited storage, and the handling of interrupts. In the event of transaction failure during the Suspended execution state, failure handling is deferred until transactional execution is resumed, allowing the block of Suspended state code to complete its activities.

--- Programming Note ---

During Suspended state execution, accessing storage locations that have been transactionally accessed by the same thread prior to entering Suspended state requires special care, because failure may occur due to uncontrollable events such as interactions with other threads or the operating system. Up until a transaction fails, loads from transactionally modified storage locations will return the transactionally modified data. However once the transaction fails, the loads may return either the transactionally updated version of storage, or a non-transactional version. Suspended state stores to transactionally modified blocks cause the thread’s transaction to fail.

Table 9 enumerates the set of Transactional Memory instructions and events that can cause changes to the transaction state. Transaction states are abbreviated N (Non-transactional), T (Transactional), and S (Suspended). (Interrupts, and the rfebb, rfid, rfscv, hrfid,
and mtmsrd instructions, can also cause changes to the transaction state; see Book III.)

--- Programming Note ---

tbegin. in Suspended state merely updates CR0. When tbegin. is followed by beq, this will result in a transfer to the failure handler. Nothing more severe (e.g. an interrupt) is required. The failure handler for a transaction for which initiation may be attempted in Suspended state should test CR0 to determine whether tbegin. was executed in Suspended state. If so, it should attempt to emulate the transaction non-transactionally. (This case can arise, for example, if a transaction enters Suspended state and then calls a library routine that independently attempts to use transactions.)

Notice that, although a failure handler runs in Non-transactional state when reached because the transaction has failed, it runs in Suspended state for the case discussed in this Programming Note.)
5.2.1 The TDOOMED Bit

The status of an active transaction is summarized by a transaction doomed bit (TDOOMED) that resides in an implementation-dependent location. When 0, it indicates that the active transaction is valid, meaning that it remains possible for the transaction to commit successfully, if failure does not occur before committing. When 1 it indicates that transaction failure has already occurred for the transaction.

The TDOOMED bit is set to 0 upon the successful initiation of an outer transaction by \textit{tbegin}. It is set to 1 when failure occurs or as a result of executing \textit{trechkpt}. When failure occurs, TDOOMED is set to 1 before any other effects of the transaction failure (recording the failure in TEXASR, rollback of transactional stores, over-writing of the transactionally accessed locations by a conflicting store, etc.) are visible to software executing on the processor that executed the transaction. In Non-transactional state, the value of TDOOMED is undefined.

5.3 Transaction Failure

5.3.1 Causes of Transaction Failure

A transaction failure is said to be “externally-induced” if the failure is caused by a thread other than the transactional thread. Likewise, a transaction failure is said to be “self-induced” if the failure is caused by the transactional thread itself.

For self-induced failure as a result of attempting to execute an instruction that is disallowed in Transactional state, or an \textit{mtspr} specifying an SPR that is not part of the checkpointed registers and is not the GSR or a Transactional Memory SPR, Privileged Instruction type Program interrupt, Hypervisor Emulation Assistance interrupt, and [Hypervisor] Facility Unavailable interrupt take precedence over transaction failure. (For example, an attempt to execute \textit{stdcx} in Transactional state and problem state will result in a Privileged Instruction type Program interrupt.) For these instructions, transaction failure takes precedence over all other interrupt types. The relevant instructions are listed in the fourth and fifth bullets of the second set of bullets below and the first and second bullets in the third set of bullets below.

In general, a ROT will not fail in the following scenarios when the failure is specified as a conflict on a transactional access and the access is a load.

Transactions will fail for the following externally-induced causes:

```plaintext
<table>
<thead>
<tr>
<th>Instr/Event State</th>
<th>tbeg.</th>
<th>tend.</th>
<th>abortcaused by tabort. and conditional tabort. variants</th>
<th>tsuspend</th>
<th>treume</th>
<th>Failure</th>
<th>treclaim</th>
<th>trechkpt</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td></td>
<td>N2</td>
<td></td>
<td>N2</td>
<td>N2</td>
<td>Notapplicable</td>
<td>N6</td>
<td>S7</td>
</tr>
<tr>
<td>T</td>
<td></td>
<td>N, if outer transaction or A=1 form; otherwise T</td>
<td>N3,4</td>
<td>S</td>
<td>T</td>
<td>N3,4</td>
<td>N3</td>
<td>S6</td>
</tr>
<tr>
<td>S</td>
<td>S1</td>
<td>S6</td>
<td>S3</td>
<td>S2</td>
<td>T5</td>
<td>S3</td>
<td>N3</td>
<td>S6</td>
</tr>
</tbody>
</table>

Notes:
1. CR0 updated indicating transactional initiation was unsuccessful, due to a pre-existing transaction occupying the transactional facility.
2. Execution of these operations does not affect transaction state, allowing for the instructions to be used in software modules called from Non-transactional, Transactional, and Suspended paths.
3. If failure recording has not previously occurred, failure recording is performed as defined in Section 5.3.2.
4. Failure handling is performed as defined in Section 5.3.3.
5. If failure has occurred during Suspended execution, failure handling will be performed sometime after the execution of \textit{tresume}, and no later than the set of events listed in Section 5.3.3.
6. Generate TM Bad Thing type Program interrupt.
7. If TEXASRFs=0, generate a TM Bad Thing type Program interrupt.

Table 9: Transaction state transitions caused by TM instructions and transaction failure
- Conflict with transactional access by another thread
- Conflict with non-transactional access by another thread
- In either of the previous two cases, if a successful Store Conditional would have conflicted, but the Store Conditional is not successful, it is implementation-dependent whether a conflict is detected
- Conflict with a translation invalidation caused by a tlbie or slbieg performed by another thread
- copy from a block that was previously written transactionally is executed by another thread.
- paste, to a block that was previously accessed transactionally is executed by another thread.
- Footprint overflow that occurs when the thread is sharing the transactional footprint tracking resources with other threads. Footprint overflow is defined as an attempt to perform a storage access in Transactional state which exceeds the capacity for tracking transactional accesses.

Transactions will fail for the following self-induced causes
- Termination caused by the execution of tabort, tabortdc, tabortdci, tabortwc, tabortwci, or treclaim instruction.
- Transaction level overflow, defined as an attempt to execute tbegin, when the transaction level is already at its maximum value
- Footprint overflow that occurs when the thread is the only thread using the transactional footprint tracking resources.
- Execution of the following instructions while in the Transactional state: icbi, copy, paste, cpabort, lwat, ldat, stwat, stdat, dcbf, dcbi, dcbst, rfscv, [h]rfid, rfobb, mtmsr[df], msgsnd, msgsndp, msgclr, msgsclp, slbie[g], slbia, slbtme, slbf, stop, and tlbie[]. (These instructions are considered to be disallowed in Transactional state.) The disallowed instruction is not executed; failure handling occurs before it has been executed.

<table>
<thead>
<tr>
<th>Programming Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>Note that execution of a stop instruction in Suspended state causes a TM Bad Thing Program interrupt.</td>
</tr>
</tbody>
</table>

- Execution, while in Transactional state, of mtspr specifying an SPR that is not part of the check-pointed registers and is not the GSR or a Transactional Memory SPR. The mtspr is not executed; failure handling occurs before it has been executed. (Modification of XERFXCC and CRCR0 are allowed, but the changes will not be rolled back in the event of transaction failure.)
- Conflict caused by a Suspended state store to a storage location that was previously accessed transactionally. If the store would have been performed by a successful Store Conditional instruction, but the Store Conditional instruction does not succeed, it is implementation-dependent whether a conflict is detected.
- Conflict caused by a Suspended state Load Atomic or Store Atomic instruction updating a block that was previously accessed transactionally.
- Conflict caused by a Suspended state tlbie or slbieg that specifies a translation that was previously used transactionally. (This case will be recorded as a translation invalidation conflict because it may be hard to differentiate from a conflict caused by a tlbie or slbieg performed by another thread and because it is highly likely to be a transient failure.)

For each of the following potential causes, the transaction will fail if the absence of failure would compromise transaction semantics; otherwise, whether the transaction fails is undefined.

- Execution of the following instructions while in the Transactional state: lbzcx, ldex, lhzcx, lwzcx, stbcx, stdcx, sthcx, stwcx. The disallowed instruction is not executed; failure handling occurs before it has been executed. (These instructions are considered to be disallowed in Transactional state if they cause transaction failure in Transactional state.) Execution of these instructions in the Suspended state is allowed and does not cause transaction failure.
- Execution of the following instruction in the Transactional state: wait. The disallowed instruction is not executed; failure handling occurs before it has been executed. (This instruction is considered to be disallowed in a transaction if it causes transaction failure.)
- Execution of the following instruction in the Suspended state: wait. The disallowed instruction is treated as a no-op; failure recording occurs. (This instruction is considered to be disallowed in a transaction if it causes transaction failure.)
- Access of a disallowed type while in the Transactional state: Caching Inhibited, Write Through Required, and Memory Coherence not Required for data access; Caching Inhibited for instruction fetch. The disallowed access is not performed; failure handling occurs such that the instruction that would cause (or be associated with, for instruction fetch) the disallowed access type appears not to have been executed. Accesses of this type in the Suspended state are allowed and do not cause transaction failure.
- Instruction fetch from a storage location that was previously written transactionally (reported as a unique cause that includes both self-induced and externally-induced instances)
- dcbf, dcbi, or icbi specifying a block that was previously accessed transactionally, in either of the following cases.
the instruction ($dcbf$, $dcbi$, or $icbi$) is executed in Suspended state on the processor executing the transaction (self-induced conflict)
- the instruction is executed by another processor (externally-induced conflict)
- $dcbst$ specifying a block that was previously written transactionally, in either of the following cases.
  - $dcbst$ is executed in Suspended state on the processor executing the transaction (self-induced conflict)
  - $dcbst$ is executed by another processor (externally-induced conflict)
- $copy$, in any of the following cases.
  - $copy$ from a block that was previously accessed transactionally is executed in Suspended state on the processor executing the transaction (self-induced conflict)
  - $copy$ from a block that was previously accessed transactionally is executed by another processor (externally-induced conflict)
- Cache eviction of a block that was previously accessed transactionally. (This case will be recorded as an externally-induced footprint overflow for several reasons. First, it is also a case in which over-use of hardware resources precludes complete tracking of the transaction. Second, eviction that is self-induced (i.e., due solely to cache use by the executing thread) may be difficult for hardware to differentiate from eviction that is due partly to cache use by other threads. Finally, this case is expected to occur only rarely, and therefore not to be worth the one or two additional TEXASR bits that would be needed to record it separately.)

Transactions may also fail due to implementation-specific characteristics of the transactional memory mechanism.

Programming Note

Warning: Software should not depend for its correct execution on the behavior (whether or not the relevant transaction fails) of the cases described in the preceding set of bullets. The behavior is likely to vary from design to design. Such a dependence would impact the software’s portability without any tangible advantage.

Programming Note

Because the atomic nature of a transaction implies an apparent delay of its component accesses until they can be performed in unison, the use of cache control instructions to manage cache residency and/or the performing of storage accesses may have unexpected consequences. Although they may not cause transaction failure directly, their use in a transaction is strongly discouraged.

If an instruction or event does not cause transaction failure, it behaves as defined in the architecture.

The set of failure causes and events are further classified as “precise” and “imprecise” failure causes. All externally induced events are imprecise, and all self-induced events are precise with the exception of the following cases:
- Self-induced conflicts caused by instruction fetch
- Self-induced conflicts caused by footprint overflow
- Self-induced conflicts in Suspended state (because failure handling is deferred in Suspended state).

When failure recording and handling occur (as defined in Section 5.3.2 and 5.3.3) for a precise failure, they will occur precisely according to the sequential execution model, adhering to the following rules:

1. Effects of the failure occur such that all instructions preceding the instruction causing the failure appear to have completed with respect to the executing thread.
2. The instruction causing the failure may appear not to have begun execution (except for causing the failure), or may have completed, depending on the failure cause.
3. Architecturally, no subsequent instruction has begun execution.

Failure handling for imprecise failure types is guaranteed to occur no later than the execution of $tend$, with $A=1$ or $TEXASR_{TL}=1$. Failure recording for imprecise failure types is guaranteed to occur no later than failure handling. Any operation that can cause imprecise failure if performed in-order can also cause imprecise failure if performed out-of-order.

Programming Note

Because instruction fetch from a transactionally modified storage location may result in transaction failure, and because conflict between storage accesses may be detected at granularity as large as a cache block, it is recommended that instructions and transactionally accessed data not be co-located within a single cache block.
5.3.2 Recording of Transaction Failure

When transaction failure occurs, information about the cause and circumstances of failure are recorded in SPRs associated with the transactional facility. Failure recording is performed a single time per transaction that fails, controlled by the state of the TEXASR failure summary (FS) bit; when 0, FS indicates that failure recording has not already been performed, and is therefore permissible.

The following RTL function specifies the actions taken during the recording of transaction failure:

```
TMRecordFailure(FailureCause)
#FailureCause is 32-bit cause
if TEXASRFS = 0
  if failure IA known then
    TFIAR ← CIA
    TEXASR37 ← 1
  else
    TFIAR ← approximate instruction address
    TEXASR37 ← 0
  TEXASR:31 ← FailureCause
  if MSRreg0b01 then TEXASR_suspended ← 1
  TEXASRPrivilege ← MSRUV PR
  TFIARPrivilege ← MSRUV PR
  TEXASRFS ← 1
  TDOOMED ← 1
```

When failure recording occurs, the TEXASR and TFIAR SPRs are set indicating the source of failure. When possible, TFIAR is set to the effective address of the instruction that caused the failure, and TEXASR37 is set to 1 indicating that the contents of TFIAR are exact. When the instruction address is not known exactly, an approximate value is placed in TFIAR and TEXASR37 is set to 0. TEXASR bits 0:31 are set indicating the cause of the failure, and the TEXASR_suspended, TEXASRPrivilege, and TFIARPrivilege fields are set indicating the machine state in which the failure was recorded. TEXASRTL is unchanged. The TDOOMED bit is set to 1.

5.3.3 Handling of Transaction Failure

Discarding of the transactional footprint may begin immediately after detection of failure and, except in the case of an abort in Suspended state, may continue until the rest of failure handling is complete. However, the timing of the rest of failure handling is dependent on the state of the transactional facility. In the case of an abort in Suspended state, the transactional footprint is discarded immediately, despite that the rest of failure handling is deferred.

In Transactional state, failure handling may occur immediately, but an implementation is free to delay handling until one of the following failure synchronizing events occurs in Transactional state:

- An abort caused by the execution of a `tabort`, `tabortdc`, `tabortdc1`, `tabortwc`, or `tabortwci` instruction.
- The execution of a `treclaim` instruction.
- An attempt, in Transactional state, to execute a disallowed instruction, perform an access of a disallowed type, or execute an `mtspr` instruction that specifies an SPR that is not part of the checkpointed registers and is not the GSR or a Transactional Memory SPR.
- Nesting level overflow.
- An attempt to transition from Transactional to Suspended state caused by `tsuspend` or by an interrupt or event.
- An attempt to commit a transaction, caused by the execution of `tend` with A = 1 or when TEXASRTL = 1.

When a failure synchronizing event occurs in Transactional state, the processor waits until all preceding Transactional and Suspended state loads have been performed with respect to all processors and mechanisms and all failures that have occurred up to that point have been recorded. Then failure handling occurs if a failure has been recorded; otherwise, processing of the failure synchronizing event continues. If failure is caused by the failure synchronizing event, failure handling occurs immediately.

When failure handling occurs, checkpointed registers are reverted to their pre-transactional values, the transactional footprint is discarded if it has not previously been discarded, and any resources occupied by the transaction are discarded. If the failure is not caused by
the following things occur. CR0 is set to 0b101 || 0. The transaction state is set to Non-transactional, and control flow is redirected to the instruction address stored in TFHAR. If the failure is caused by `treclaim`., CR0 is not set to indicate failure and the transaction’s failure handler is not invoked.

The following RTL function specifies the actions taken during the handling of transaction failure:

```
TMHandleFailure()
    If the transactional footprint has not previously been discarded
        Discard transactional footprint
        Revert checkpointed registers to pre-transactional values
        Discard all resources related to current transaction
        MSRset <- 0b00 #Non-transactional
        If failure was not caused by `treclaim`.,
        NIA = TFHAR
        CR0 = 0b101 || 0

Upon failure detected in Suspended state from causes other than the execution of a `treclaim` instruction, failure handling is deferred until the transaction is resumed. Once resumed, failure handling will occur no later than the set of failure synchronizing events listed above. Upon failure in Suspended state caused by `treclaim`, failure handling is immediate (but CR0 is not set to indicate failure and the transaction’s failure handler is not invoked).
```

5.4.1 Transaction Failure Handler Address Register (TFHAR)

The Transaction Failure Handler Address Register is a 64-bit SPR that records the effective address of a software failure handler used in the event of transaction failure. Bits 62:63 are reserved.

![Figure 5. Transaction Failure Handler Address Register (TFHAR)](image)

This register is written with the NIA for the `tbegin` as a side-effect of the execution of an outer `tbegin` instruction (`tbegin` executed in the Non-transactional state).

5.4.2 Transaction EXception And Status Register (TEXASR)

The Transaction EXception And Status Register is a 64-bit register, containing a transaction level (TEXASR) and status information for use by transaction failure handlers. The identification of the cause and persistence of transaction failure reported in bits 7:30 may rarely be inaccurate, except that if bit 31 is set to 1 then bits 7:30 are always accurate. Bits 0:31 are called the failure cause in the instruction descriptions.

```
Programming Note

A Load instruction executed immediately after `treclaim` or a conditional or unconditional Abort instruction is guaranteed not to load a transactional storage update.
```

5.4 Transactional Memory Facility Registers

The architecture is augmented with three Special Purpose Registers in support of transactional memory. TFHAR stores the effective address of the software failure handler used in the event of transaction failure. TFIAR is used to inform software of the exact location of the transaction failure, when possible. TEXASR contains a transaction level indicating the nesting depth of an active transaction, as well as an indicator of the cause of transaction failure and some machine state when the transaction failed. These registers can be written only when in Non-transactional state, and for TFHAR, also when in Suspended state.

**Programming Note**

Warning: In addition to the contents of bits 7:30 of the TEXASR being rarely inaccurate, new failure causes may be added over time, and/or an existing cause may be divided into two or more subsets of causes (which may differ in their persistence). Further, speculative execution can cause unexpected contents of these bits. As a result, except when failure is caused by a `treclaim` or `tabort` instruction (including conditional `tabort`), software must not depend on the contents of bits 7:30 for its correct execution. Guidelines follow.

- The bits should only be used to determine the approach to the computation, e.g., whether the computation is suitable to use the Transactional Memory facility, how to adapt it best for TM, or how many retries to attempt before performing the operation non-transactionally.
- Software should use the persistence indication when none of the causes that were defined at the time the software was written is indicated.
- Under no circumstances should software depend on the transience of a failure. There must always be a limit to the number of retries before performing the operation non-transactionally.
Figure 6. Transaction EXception And Status Register (TEXASR)

Figure 7. Transaction EXception And Status Register Upper (TEXASRU)

<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:6</td>
<td>Failure Code</td>
</tr>
<tr>
<td></td>
<td>The Failure Code is copied from the tabort, or treclaim, source operand. When set, TFIAR is exact.</td>
</tr>
<tr>
<td>7</td>
<td>Failure Persistent</td>
</tr>
<tr>
<td></td>
<td>The failure is likely to recur on each execution of the transaction. This bit is set to 1 for causes in bits 8:11, copied from the tabort, or treclaim, source operand when RA is non-zero, and set to 0 for all other failure causes.</td>
</tr>
<tr>
<td>8</td>
<td>Disallowed</td>
</tr>
<tr>
<td></td>
<td>The instruction, SPR, or access type is not permitted. When set, TFIAR is exact. See Section 5.3.1, “Causes of Transaction Failure”.</td>
</tr>
<tr>
<td>9</td>
<td>Nesting Overflow</td>
</tr>
<tr>
<td></td>
<td>The maximum transaction level was exceeded. When set, TFIAR is exact.</td>
</tr>
<tr>
<td>10</td>
<td>Footprint Overflow, Self-Induced</td>
</tr>
<tr>
<td></td>
<td>The tracking limit for transactional storage accesses was exceeded when this thread was the only thread using the transactional footprint tracking resources. When set, TFIAR is an approximation.</td>
</tr>
<tr>
<td>11</td>
<td>Self-Induced Conflict</td>
</tr>
<tr>
<td></td>
<td>A self-induced conflict occurred in Suspended state, due to one of the following: a store to a storage location that was previously accessed transactionally; a <em>dcbf, dcbi, or icbi</em> specifying a block that was previously accessed transactionally; a <em>dcbst</em> specifying a block that was previously written transactionally; a <em>Load Atomic</em> or <em>Store Atomic</em> instruction specifying a block that was previously accessed transactionally, or a <em>copy</em> from a block that was previously accessed transactionally. When set, TFIAR may be exact.</td>
</tr>
<tr>
<td>12</td>
<td>Non-Transactional Conflict</td>
</tr>
<tr>
<td></td>
<td>A conflict occurred with a non-transactional access by another processor. When set, TFIAR is an approximation.</td>
</tr>
<tr>
<td>13</td>
<td>Transaction Conflict</td>
</tr>
<tr>
<td></td>
<td>A conflict occurred with another transaction. When set, TFIAR may be exact.</td>
</tr>
<tr>
<td>14</td>
<td>Translation Invalidation Conflict</td>
</tr>
<tr>
<td></td>
<td>A conflict occurred with a TLB or SLB invalidation. When set, TFIAR is an approximation.</td>
</tr>
<tr>
<td>15</td>
<td>Implementation-specific</td>
</tr>
<tr>
<td></td>
<td>An implementation-specific condition caused the transaction to fail. Such conditions are transient and the value in the TFIAR may be exact.</td>
</tr>
<tr>
<td>16</td>
<td>Instruction Fetch Conflict</td>
</tr>
<tr>
<td></td>
<td>An instruction fetch (by this or another thread) was performed from a storage location that Caching Inhibited. The inaccuracy of the Failure Persistent bit may arise from either of two causes. First, a kind of failure that is usually transient, such as conflict with another thread, may in certain unusual circumstances be persistent. Second, if the cause of transaction failure is identified incorrectly, the Failure Persistent bit will inherit this inaccuracy -- i.e., will be set to 0 or 1 based on the identified failure cause. (Neither of these causes applies if TEXASR_{31}=1.)</td>
</tr>
</tbody>
</table>

Programming Note

For *tabort* and *treclaim*, the Failure Persistent bit may be viewed as an eighth bit in the failure code in that both fields are supplied by the least significant byte of RA and software may use all eight to differentiate among the cases for which it performs an abort or reclaim. However, software is expected to organize its cases so that bit 7 predicts the persistence of the case.

Programming Note

An instruction fetch to storage that is Caching Inhibited, while nominally disallowed, will be reported as Implementation-specific (bit 15). This choice was made because it seems like a relatively unlikely programming error, and there is a significant chance that data from an external conflict (store by another thread) could indirectly cause a wild branch to storage that is Caching Inhibited.
was previously written transactionally. Such conditions are transient and the value in the TFIAR may be exact.

17 Footprint Overflow, Externally-Induced
The tracking limit for transactional storage accesses was exceeded when other threads, in addition to this thread, were using the transactional footprint tracking resources. This bit is also set when a cache block eviction causes the transaction to fail. When set, TFIAR is an approximation.

Programming Note
Appropriate behavior of the failure handler when the tracking limit is exceeded due partly to transactions running on other threads may include re-executing the transaction after a significant and randomized amount of time has elapsed. (This policy will tend to spread out the contending transactions in time, and thereby reduce their simultaneous use of the transactional footprint tracking resources.) Some designs may provide hardware assistance in reducing contention for the tracking resources. Writers of failure handlers should see the Users' Manual for the implementation to understand how to benefit from the hardware behavior.

Transaction failure due to cache block eviction is expected to be sufficiently rare that handling it as if the failure were caused by exceeding the tracking limit is acceptable.

18:30 Reserved for future failure causes
31 Abort
Termination was caused by the execution of a \textit{tabort}, \textit{tabortdc}, \textit{tabortdc1}, \textit{tabortwc}, \textit{tabortwc1} or \textit{treclaim} instruction. When due to \textit{tabort} or \textit{treclaim}, bits in TEXASR0:7 are user-supplied. When set, TFIAR is exact.

32 Suspended
When set to 1, the failure was recorded in Suspended state. When set to 0, the failure was recorded in Transactional state.

33 Reserved

34:35 Privilege
The thread was in this privilege state when the failure was recorded. This was the value MSR$_{HV}$ or MSR$_{PR}$ when the failure was recorded.

36 Failure Summary (FS)
Set to 1 when a failure has been detected and failure recording has been performed.

37 TFIAR Exact
Set to 1 when the value in the TFIAR is exact. Otherwise the value in the TFIAR is approximate.

38 ROT
Set to 1 when a ROT is initiated. Set to zero when a non-ROT \textit{tbegin.} is executed.

39 Reserved

40:51 Reserved

52:63 Transaction Level (TL)
Transaction level (nesting depth + 1) for the active transaction, if any; otherwise 0 if the most recently executed transaction completed successfully, or the transaction level at which the most recently executed transaction failed if the most recently executed transaction did not complete successfully.

The transaction level in TEXASR$_{TL}$ contains an unsigned integer indicating whether the current transaction is an outer transaction (TEXASR$_{TL} = 1$), or is nested (TEXASR$_{TL} > 1$), and if nested, its depth. The maximum transaction level supported by a given implementation is of the form $2^t - 1$. The value of $t$ corresponding to the smallest maximum is 4; the value of $t$ corresponding to the largest maximum is 12. This value is tied to the “Maximum transaction level” parameter useful for application programmers, as specified in Section 4.1. The high-order 12-$t$ bits of TEXASR$_{TL}$ are treated as reserved.

Transaction failure information is contained in TEXASR$_{0:37}$. The fields of TEXASR are initialized upon the successful initiation of a transaction from the Non-transactional state, by setting TEXASR$_{TL}$ to 1, indicating an outer transaction, and all other fields to 0.

When transaction failure is recorded, the failure summary bit TEXASR$_{FS}$ is set to 1, indicating that failure has been detected for the active transaction and that failure recording has been performed. TEXASR$_{0:31}$ are set indicating the source of the failure. Exactly one of bits 8 through 31 will be set indicating the instruction or event that caused failure. In the event of failure due to the execution of a \textit{tabort}, \textit{tabortdc}, \textit{tabortdc1}, \textit{tabortwc}, \textit{tabortwc1}, \textit{treclaim} instruction, TEXASR$_{31}$ is set to 1, and, for \textit{tabort} and \textit{treclaim}, a software defined failure code is copied from a register operand to TEXASR$_{0:7}$. TEXASR$_{Suspended}$ indicates whether the transaction was in the Suspended state at the time that failure occurred. The values of MSR$_{HV}$ and MSR$_{PR}$ at the time that failure occurs are copied to TEXASR$_{34}$ and TEXASR$_{35}$, respectively. In some circumstances, the failure causing instruction address in TFIAR may not be exact. In such circumstances, TEXASR$_{37}$ is set to 0 indicating that the contents of TFIAR are not exact; otherwise TEXASR$_{37}$ is set to 1.
5.4.3 Transaction Failure Instruction Address Register (TFIAR)

The Transaction Failure Instruction Address Register is a 64-bit SPR that is set to the exact effective address of the instruction causing the failure, when possible. Bits 62:63 contain the privilege state when the failure was recorded. This was the value MSR_{HV\ PR} when the failure was recorded.

<table>
<thead>
<tr>
<th>TFIA</th>
<th>Privilege</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>62</td>
</tr>
<tr>
<td></td>
<td>63</td>
</tr>
</tbody>
</table>

In certain cases, the exact address may not be available, and therefore TFIAR will be an approximation. An approximate value will point to an instruction near the instruction that was executing at the time of the failure. TFIAR accuracy is recorded in an Exact bit residing in TEXASR_{37}.

Programming Note
The transaction level contained in TEXASRTL should be interpreted by software as follows:

When in the Transactional or Suspended state, this field contains an unsigned integer representing the transaction level of the active transaction, with 1 indicating an outer transaction, and a number greater than 1 indicating a nested transaction. The nesting depth of the active transaction is TEXASRTL - 1.

When in the Non-transactional state, TEXASRTL contains 0 if the last transaction committed successfully, otherwise it contains the transaction level at which the most recent transaction failed.

Programming Note
The Privilege bits in TEXASR represent the state of the machine at the point when failure occurs. This information may be used by problem state software to determine whether an unexpected hypervisor or operating system interaction was responsible for transaction failure. This information may be useful to operating systems or hypervisors when restoring register state for failure handling after the transactional facility was reclaimed, to determine which of the operating system or the hypervisor has retained the pre-transactional version of the checkpointed registers.
5.5 Transactional Facility Instructions

Similar to the Floating-Point Status and Control Register instructions, modifications of transaction state caused by the execution of Transactional Memory instructions or by failure handling synchronize the effects of exception-causing floating-point instructions executed by a given processor. Executing a Transactional Memory instruction, or invocation of the failure handler, ensures that all floating-point instructions previously initiated by the given processor have completed before the transaction state is modified, and that no subsequent floating-point instructions are initiated by the given processor until the transaction state has been modified. In particular:

- All exceptions that will be caused by the previously initiated instructions are recorded in the FPSCR before the transaction state is modified.
- All invocations of the system floating-point enabled exception error handler that will be caused by the previously initiated instructions have occurred before the transaction state is modified.
- No subsequent floating-point instruction that alters the settings of any FPSCR bits is initiated until the transaction state has been modified.

(Floating-point Storage Access instructions are not affected.)

### Transaction Begin X-form

<table>
<thead>
<tr>
<th>tbegin.</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>31 A // R /// // // 654 1</td>
<td></td>
</tr>
<tr>
<td>0 7 0 11 16 21 31</td>
<td></td>
</tr>
</tbody>
</table>

- **ROT** ← R
- **CR0** ← 0 || **MSR{reg}** || 0
- if **MSR{reg}** = 0b0 then  #Non-transactional
  - **TEXASR** ← 0x000000000 || 0b00 || **ROT** || 0b0 || 0x0000001
  - **TFHAR** ← CIA + 4
  - **TDOOMED** ← 0
  - **MSR{reg}** ← 0b10
  - checkpoint area ← (checkpointed registers)
  - if not ROT and the transaction succeeds then insert tbegin memory barrier
- else if **MSR{reg}** = 0b10 then  #Transactionnal
  - if **TEXASR{TL}**=TL{max} then
    - cause ← 0x01400000
    - **TMRecordFailure**(cause)
    - **TMHandleFailure**()
- else
  - **TEXASR{TL}** ← **TEXASR{TL}** + 1
  - if (**TEXASR{ROT}=1**) & (not **ROT**) **TEXASR{ROT}** ← 0
  - if the transaction succeeds insert tbegin memory barrier

The **tbegin.** instruction initiates execution of a transaction, either an outer transaction or a nested transaction, as described below.

An outer transaction is initiated when **tbegin.** is executed in the Non-transactional state. If **R=0** and the transaction is successful, the **tbegin** memory barrier described in Section 1.8 is inserted. **TEXASR** and **TFHAR** are initialized, and the **TDOOMED** bit is set to 0. A nested transaction is initiated when **tbegin.** is executed in the Transactional state unless the transaction level is already at its maximum value, in which case failure recording is performed with a failure cause of 0x01400000 and failure handling is performed. When initiating a nested transaction, the transaction level held in **TEXASR{TL}** is incremented by 1, and if **TEXASR{ROT}=1** but **R=0**, **TEXASR{ROT}** is set to 0, and if additionally the transaction succeeds, the **tbegin** memory barrier described in Section 1.8 is inserted. The effects of a nested transaction will not be visible until the outer transaction commits, and in the event of failure, the checkpointed registers are reverted to the pre-transactional values of the outer transaction. Initiation of a transaction is unsuccessful when in the Suspended state.

When successfully initiated, transactional execution continues until the transaction is terminated using a **tend.**, **tabort.**, **tabortdc.**, **tabortwc.**, **tabortwci.**, or **treclaim.** instruction, suspended using a **tsr** instruction, or failure occurs. Upon transaction failure while in the Transactional state, transaction failure recording and failure handling are performed as defined in Section 5.3. Upon transaction failure while in the Suspended state, failure recording is performed as defined in Section 5.3.2, but failure handling is usually deferred.

CR0 is set as follows.

<table>
<thead>
<tr>
<th>CR0</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td></td>
</tr>
<tr>
<td>010</td>
<td></td>
</tr>
<tr>
<td>001</td>
<td></td>
</tr>
</tbody>
</table>

Other than the setting of CR0, **tbegin.** in the Suspended state is treated as a no-op.
The use of the A field is implementation specific.

Special Registers Altered:
CR0 TEXASR TFHAR TS

--- Programming Note ---
When a transaction is successfully initiated, and failure subsequently occurs, control flow will be redirected to the instruction following the `tbegin` instruction. When failure handling occurs, as described in Section 5.3.3, CR0 is set to `0b101 || 0`. Consequently, instructions following `tbegin`, should also expect this value as an indication of transaction failure. Most applications will follow `tbegin` with a conditional branch predicated on CR0, code at this target is responsible for handling the transaction failure.

### Transaction End X-form

<table>
<thead>
<tr>
<th>A</th>
<th>X-form</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>0b00011011001111111111111111111111</td>
</tr>
</tbody>
</table>

The A=0 variant of `tend` supports nested transactions, in which the transaction is committed only if the execution of `tend` completes an outer transaction. Execution of this variant by a nested transaction (TEXASRTL > 1) causes TEXASRTL to be decremented by 1. The A=1 variant of `tend` unconditionally completes the current outer transaction and all nested transactions.

When the `tend` instruction completes an outer transaction, transaction commit is predicated on the TDOOMED bit. If TDOOMED is 1, failure handling occurs as defined in Section 5.3.3. If TDOOMED is 0, the transaction is committed, and TEXASRTL is set to 0. In both cases, the transaction state is set to Non-transactional.

When the `tend` instruction commits a transaction, it atomically commits its writes to storage. If TEXASRROT=0, the integrated cumulative memory barrier is inserted prior to the creation of the aggregate store, and the `tend` memory barrier described in Section 1.8 is inserted after the aggregate store. If the transaction has failed prior to the execution of `tend`, no storage updates are performed and no memory barrier is inserted. In either case (success or failure), all resources associated with the transaction are discarded.

If the transaction succeeds, Condition Register field 0 is set to `0 || MSRTS || 0`. If the transaction fails, CR0 is set to `0b101 || 0`.

Other than the setting of CR0, `tend` in Non-transactional state is treated as a no-op. If an attempt is made to execute `tend` in Suspended state, a TM Bad Thing type Program interrupt occurs.
Special Registers Altered:
  CR0 TEXASR TFIAR TS

Extended Mnemonics
Examples of extended mnemonics for Transaction End.

<table>
<thead>
<tr>
<th>Extended:</th>
<th>Equivalent To:</th>
</tr>
</thead>
<tbody>
<tr>
<td>tend.</td>
<td>tend. 0</td>
</tr>
<tr>
<td>tendall.</td>
<td>tend. 1</td>
</tr>
</tbody>
</table>

Programming Note
When an outer tend. or a tend. with A=1 is executed in the Transactional state, the CR0 value 0b101 || 0 will never be visible to the instruction that immediately follows tend., because in the event of failure the failure handler will have been invoked not later than the completion of the tend. instruction.

Transaction Abort

<table>
<thead>
<tr>
<th>X-form</th>
<th>RA</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>///</td>
</tr>
<tr>
<td>0</td>
<td>///</td>
</tr>
<tr>
<td>6</td>
<td>///</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
CR0 &\leftarrow 0 \ || \ MSR_{TS} \ || \ 0 \\
\text{if} \ MSR_{TS} &= 0b10 \ | \ MSR_{TS} = 0b01 \ \text{then} \\
&\#\text{Transactional, or Suspended} \\
&\text{if} \ RA = 0 \ \text{then} \ cause \leftarrow \text{GPR(RA)}_{56:63} \ || \ 0x00000001 \\
&\text{else} \ \text{cause} \leftarrow \text{GPR(RA)}_{56:63} \ || \ 0x00000001 \\
&\text{if} \ MSR_{TS} = 0b01 \ & \text{TExASR}_{FS} = 0 \ \text{then} \ #\text{Suspended} \\
&\text{Discard the transactional footprint} \\
&\text{TMRecordFailure(cause)} \\
&\text{if} \ MSR_{TS} = 0b10 \ \text{then} \ #\text{Transactional} \\
&\text{TMHandleFailure()} \\
\end{align*}
\]

The tabort. instruction sets condition register field 0 to 0 || MSR_{TS} || 0. When in the Transactional state or the Suspended state the tabort. instruction causes transaction failure, resulting in the following:

Failure recording is performed as defined in Section 5.3.2. If RA is 0, the failure cause is set to 0x00000001, otherwise it is set to GPR(RA)_{56:63} || 0x00000001.

If the transaction state is Transactional, failure handling is performed as defined in Section 5.3.3 (this includes discarding the transactional footprint).

If the transaction state is Suspended, the transactional footprint is discarded (if not already discarded for a pending failure), but failure handling is deferred.

Other than the setting of CR0, execution of tabort. in the Non-transactional state is treated as a no-op.

Special Registers Altered:
  CR0 TEXASR TFIAR TS
**Transaction Abort Word Conditional**

**X-form**

```
tabortwc. TO,RA,RB
```

<table>
<thead>
<tr>
<th>31</th>
<th>8</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>782</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```
a ← EXTS((RA)32:63) 
b ← EXTS((RB)32:63) 
abort ← 0 

CR0 ← 0 || MSR_TS || 0 

if (a < b) & TO0 then abort ← 1 
if (a > b) & TO1 then abort ← 1 
if (a = b) & TO2 then abort ← 1 
if (a u< b) & TO3 then abort ← 1 
if (a >u b) & TO4 then abort ← 1 

if abort & (MSRTS = 0b10 | MSRTS = 0b01) then 
  #Transactional or Suspended 
  cause ← 0x00000001 
  if MSRTS= 0b01 & TEXASRFS = 0 then  #Suspended 
    Discard transactional footprint 
    TMRecordFailure(cause) 
  if MSRTS = 0b10 then                #Transactional 
    TMHandleFailure() 

The *tabortwc*. instruction sets condition register field 0 to 0 || MSR_TS || 0. The contents of register RA_{32:63} are compared with the contents of register RB_{32:63}. If any bit in the TO field is set to 1 and its corresponding condition is met by the result of the comparison, and the transaction state is Transactional or Suspended, then the *tabortwc*. instruction causes transaction failure, resulting in the following:

Failure recording is performed as defined in Section 5.3.2, using the failure cause 0x00000001.

If the transaction state is Transactional, failure handling is performed as defined in Section 5.3.3 (this includes discarding the transactional footprint).

If the transaction state is Suspended, the transactional footprint is discarded (if not already discarded for a pending failure), but failure handling is deferred.

Other than the setting of CR0, execution of *tabortwc*. in the Non-transactional state is treated as a no-op.

**Special Registers Altered:**

CR0 TEXASR TFIAR TS

---

**Transaction Abort Word Conditional**

**Immediate X-form**

```
tabortwci. TO,RA,SI
```

<table>
<thead>
<tr>
<th>31</th>
<th>8</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>846</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```
a ← EXTS((RA)32:63) 
abort ← 0 

CR0 ← 0 || MSR_TS || 0 

if a < EXTS(SI) & TO0 then abort ← 1 
if a > EXTS(SI) & TO1 then abort ← 1 
if a = EXTS(SI) & TO2 then abort ← 1 
if a u< EXTS(SI) & TO3 then abort ← 1 
if a >u EXTS(SI) & TO4 then abort ← 1 

if abort & (MSRTS = 0b10 | MSRTS = 0b01) then 
  #Transactional or Suspended 
  cause ← 0x00000001 
  if MSRTS= 0b01 & TEXASRFS = 0 then  #Suspended 
    Discard transactional footprint 
    TMRecordFailure(cause) 
  if MSRTS = 0b10 then                #Transactional 
    TMHandleFailure() 

The *tabortwci*. instruction sets condition register field 0 to 0 || MSR_TS || 0. The contents of register RA_{32:63} 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, and the transaction state is Transactional or Suspended then the *tabortwci*. instruction causes transaction failure, resulting in the following:

Failure recording is performed as defined in Section 5.3.2, using the failure cause 0x00000001.

If the transaction state is Transactional, failure handling is performed as defined in Section 5.3.3 (this includes discarding the transactional footprint).

If the transaction state is Suspended, the transactional footprint is discarded (if not already discarded for a pending failure), but failure handling is deferred.

Other than the setting of CR0, execution of *tabortwci*. in the Non-transactional state is treated as a no-op.

**Special Registers Altered:**

CR0 TEXASR TFIAR TS

---

Chapter 5. Transactional Memory Facility 893
**Transaction Abort Doubleword Conditional X-form**

**tabortdc.**  TO, RA, RB

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>8</th>
<th>16</th>
<th>21</th>
<th>814</th>
<th>1</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>←</td>
<td>( RA )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>←</td>
<td>( RB )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>abort</td>
<td>←</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CR0</td>
<td>←</td>
<td>0</td>
<td></td>
<td>MSR_RS</td>
<td></td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

if (a < b) & TO0 then abort ← 1
if (a > b) & TO1 then abort ← 1
if (a = b) & TO2 then abort ← 1
if (a <u b) & TO3 then abort ← 1
if (a >u b) & TO4 then abort ← 1
if abort & (MSR_RS = 0b10 | MSR_RS = 0b01) then

#Transactional or Suspended

cause ← 0x00000001
if MSR_RS = 0b01 & TEXASR_RS = 0 then #Suspended
Discard transactional footprint
TMRecordFailure(cause)
if MSR_RS = 0b10 then #Transactional
TMHandleFailure()

The **tabortdc.** instruction sets condition register field 0 to 0 || MSR_RS || 0. 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, and the transaction state is Transactional or Suspended, then the **tabortdc.** instruction causes transaction failure, resulting in the following:

Failure recording is performed as defined in Section 5.3.2, using the failure cause 0x00000001.

If the transaction state is Transactional, failure handling is performed as defined in Section 5.3.3 (this includes discarding the transactional footprint).

If the transaction state is Suspended, the transactional footprint is discarded (if not already discarded for a pending failure), but failure handling is deferred.

Other than the setting of CR0, execution of **tabortdc.** in the Non-transactional state is treated as a no-op.

**Special Registers Altered:**

CR0 TEXASR TFIAR TS

---

**Transaction Abort Doubleword Conditional Immediate X-form**

**tabortdci.**  TO, RA, SI

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>8</th>
<th>16</th>
<th>21</th>
<th>878</th>
<th>1</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>←</td>
<td>( RA )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>abort</td>
<td>←</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CR0</td>
<td>←</td>
<td>0</td>
<td></td>
<td>MSR_RS</td>
<td></td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

if a < EXTS(SI) & TO0 then abort ← 1
if a > EXTS(SI) & TO1 then abort ← 1
if a = EXTS(SI) & TO2 then abort ← 1
if a <u EXTS(SI) & TO3 then abort ← 1
if a >u EXTS(SI) & TO4 then abort ← 1
if abort & (MSR_RS = 0b10 | MSR_RS = 0b01) then

#Transactional or Suspended

cause ← 0x00000001
if MSR_RS = 0b01 & TEXASR_RS = 0 then #Suspended
Discard transactional footprint
TMRecordFailure(cause)
if MSR_RS = 0b10 then #Transactional
TMHandleFailure()

The **tabortdci.** instruction sets condition register field 0 to 0 || MSR_RS || 0. 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, and the transaction state is Transactional or Suspended then the **tabortdci.** instruction causes transaction failure, resulting in the following:

Failure recording is performed as defined in Section 5.3.2, using the failure cause 0x00000001.

If the transaction state is Transactional, failure handling is performed as defined in Section 5.3.3 (this includes discarding the transactional footprint).

If the transaction state is Suspended, the transactional footprint is discarded (if not already discarded for a pending failure), but failure handling is deferred.

Other than the setting of CR0, execution of **tabortdci.** in the Non-transactional state is treated as a no-op.

**Special Registers Altered:**

CR0 TEXASR TFIAR TS

---

894  Power ISA™ II
Chapter 5. Transactional Memory Facility

Transaction Suspend or Resume X-form

<table>
<thead>
<tr>
<th>tsr.</th>
<th>L</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>31 6 10 11 16 21 750 1</td>
</tr>
</tbody>
</table>

CR0 ← 0 || MSR_7S || 0
if L = 0 then
  if MSR_7S = 0b10 then               #Transactional
    MSR_7S ← 0b01                   #Suspended
  else
    if MSR_7S = 0b01                    #Suspended
      MSR_7S ← 0b10                   #Transactional

The tsr. instruction sets condition register field 0 to 0 || MSR_7S || 0. Based on the value of the L field, two variants of tsr. are used to change the transaction state.

If L = 0, and the transaction state is Transactional, the transaction state is set to Suspended.

If L = 1, and the transaction state is Suspended, the transaction state is set to Transactional.

Other than the setting of CR0, the execution of tsr. in the Non-transactional state is treated as a no-op.

Special Registers Altered:
CR0 TS

Programming Note
When resuming a transaction that has encountered failure while in the Suspended state, failure handling is performed after the execution of tsr., and no later than the next failure synchronizing event.

Extended Mnemonics
Examples of extended mnemonics for Transaction Suspend or Resume.

Extended: Equivalent To:
tsuspend. tsr. 0
tresume. tsr. 1

Transaction Check X-form

tcheck BF

<table>
<thead>
<tr>
<th>tcheck</th>
<th>BF</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>31 8 9 11 16 21 718</td>
</tr>
</tbody>
</table>

if MSR_7S = 0b10 || MSR_7S = 0b01 then #Transactional
  or Suspended
  for each load caused by an instruction following the outer tbegin and preceding this tcheck
  if (Load instruction was executed in T state with TEXASR_ROT=0 or accessing a location previously stored transactionally) | (Load instruction was executed in S state with TEXASR_ROT=0 and accessed a location previously accessed transactionally) |
    (Load instruction was executed in S state with TEXASR_ROT=1 and accessed a location previously stored transactionally) |
    then wait until load has been performed with respect to all processors and mechanisms
CR field BF ← TDOOMED || MSR_7S || 0

If the transaction state is Transactional or Suspended, the tcheck instruction ensures that all loads that are caused by instructions that follow the outer tbegin instruction and precede the tcheck instruction and satisfy one of the following properties, have been performed with respect to all processors and mechanisms.

- The load is caused by an instruction that was executed in Transactional state, either while TEXASR_ROT=0 or accessing a location previously stored transactionally.
- The load is caused by an instruction that was executed in Suspended state while TEXASR_ROT=0 and accesses a location that was accessed transactionally.
- The load is caused by an instruction that was executed in Suspended state while TEXASR_ROT=1 and accesses a location that was stored transactionally.

The tcheck instruction then copies the TDOOMED bit into bit 0 of CR field BF, copies MSR_7S to bits 1:2 of CR field BF, and sets bit 3 of CR field BF to 0.

Other than the setting of CR field BF, execution of tcheck in the Non-transactional state is treated as a no-op.

Special Registers Altered:
CR field BF
One use of the `tcheck` instruction in Suspended state is to determine whether preceding loads from transactionally modified locations have returned the data the transaction stored. (If the transaction has failed, some of the loads may have returned a more recent value that was stored by a conflicting store, or may have returned the pre-transaction contents of the location.). It is important to use `tcheck` between any Suspended state loads that might access transactionally modified locations and subsequent computation using the Suspended-state-loaded data. Otherwise, corrupt data could cause problems such as wild branches or infinite loops.

Another use of `tcheck` in Suspended state is to determine whether the contents of storage, as seen in Suspended state, are consistent with the transaction succeeding -- e.g., whether no location that has been accessed transactionally (stored transactionally, for ROTs), and has been seen in Suspended state, has been subject to a conflict thus far. (A location is seen in Suspended state either by being loaded in Suspended state or by being loaded in Transactional state and the value (or a value derived therefrom) passed, in a register, into Suspended state.)

A use of `tcheck` in Transactional state is to determine whether the transaction still has the potential to succeed.

Note that `tcheck` provides an instantaneous check on the integrity of a subset of the accesses performed within a transaction. `tcheck` is not a failure synchronizing mechanism. Even if no accesses follow the `tcheck`, there may still be latent failures that haven’t been recorded, for example caused by accesses that `tcheck` does not wait for, by external conflicts that will happen in the future, or simply by time of flight to the failure detection mechanism for operations that have already been performed.

The `tcheck` instruction can return 1 in bit 0 of CR field BF before the failure has been recorded in TEXASR and TFIAR.

The `tcheck` instruction may cause pipeline synchronization. As a result, programs that use `tcheck` excessively may perform poorly.
Chapter 6. Time Base

The Time Base (TB) is a 64-bit register (see Figure 9) 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 9. Time Base

The Time Base monotonically increments until its value becomes $0xFFF_FFFF_FFFF_FFFF$ ($2^{64} - 1$); 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.
6.1 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 3.3.17 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>mftb Rx,269</td>
</tr>
<tr>
<td></td>
<td>mfspr Rx,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

The mfspr instruction can be used to read the Time Base on all processors that comply with Version 2.01 of the architecture or with any subsequent version.

It is believed that the mfspr instruction can be used to read the Time Base on most processors that comply with versions of the architecture that precede Version 2.01. Processors for which mfspr cannot be used to read the Time Base include the following.

- 601
- POWER3

(601 implements neither the Time Base nor mftb, but depends on software using mftb to read the Time Base, so that the attempt causes the Illegal Instruction error handler to be invoked and thereby permits the operating system to emulate the Time Base.)
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:

```
mfspr Ry,268  # Ry = Time Base
lwz  Rx,ticks_per_sec
divudu Rz,Ry,Rx  # Rz = whole seconds
stw  Rz,posix_sec
mulld Rz,Rz,Rx  # Rz = quotient * divisor
sub  Rz,Ry,Rz  # Rz = excess ticks
lwz  Rz,ns_adj
slwi Rz,Rz,1  # Rz = 2 * excess ticks
mulhwu Rz,Rz,Rx  # mul by (ns/tick)/2 * 2^{32}
stw  Rz,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 7. Event-Based Branch Facility

7.1 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 9.5 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.
  - The TS field of the BESCR is set to indicate the transaction state of the processor when the event-based branch occurred; if the processor was in Transactional state when the event-based branch occurred, it is put into Suspended state.
  - 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. The `rfebb` instruction also restores the processor to the transaction state indicated by BESCR\textsubscript{TS}. See the Programming Notes in Section 7.3 for an example sequence of operations of the event-based branch handler.

Additional information about the Event-Based Branch facility is given in Section 3.4 of Book III.

---

**Programming Note**

Since system software controls the availability of the Event-Based Branch facility (see Section 9.5 of Book III), an interface must be provided that enables applications to request access to the facility and determine when it is available.
7.2 Event-Based Branch Registers

7.2.1 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.

- Software requests control of the Event-Based Branch facility from the system software.
- Software requests the system software to initialize the Performance Monitor as desired.
- Software sets the EBBHR to the effective address of the event-based branch handler.
- Software enables Performance Monitor event-based exceptions by setting BESCREDITR = 1 0, and also sets MMCR0 exception enable (PMEO) = 1 0. See Section 9.4.4 of Book III for the description of MMCR0.
- Software sets the GE bit in the BESCR to enable event-based branches.

In order to initialize the Event-Based Branch facility for Performance Monitor event-based exceptions, software performs the following operations.

- Software requests control of the Event-Based Branch facility from the system software.
- Software requests the system software to initialize the Performance Monitor as desired.
- Software sets the EBBHR to the effective address of the event-based branch handler.
- Software enables Performance Monitor event-based exceptions by setting BESCREDITR = 1 0, and also sets MMCR0 exception enable (PMEO) = 1 0. See Section 9.4.4 of Book III for the description of MMCR0.
- Software sets the GE bit in the BESCR to enable event-based branches.

When $mfspr$ indicates any of the above SPR numbers, the current value of the register is returned.

In order to initialize the Event-Based Branch facility for Performance Monitor event-based exceptions, software performs the following operations.

- Software requests control of the Event-Based Branch facility from the system software.
- Software requests the system software to initialize the Performance Monitor as desired.
- Software sets the EBBHR to the effective address of the event-based branch handler.
- Software enables Performance Monitor event-based exceptions by setting BESCREDITR = 1 0, and also sets MMCR0 exception enable (PMEO) = 1 0. See Section 9.4.4 of Book III for the description of MMCR0.
- Software sets the GE bit in the BESCR to enable event-based branches.

System software controls whether or not event-based branches occur regardless of the contents of the BESCR. See Section 9.4.4 of Book III and Section 6.2.12 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.

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.

### 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.

<table>
<thead>
<tr>
<th>GE</th>
<th>Event Control</th>
<th>TS</th>
<th>Event Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>32</td>
<td>63</td>
</tr>
</tbody>
</table>

**Figure 10. Branch Event Status and Control Register (BESCR)**

<table>
<thead>
<tr>
<th>GE</th>
<th>Event Control</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

**Figure 11. Branch Event Status and Control Register Upper (BESCRU)**

System software controls whether or not event-based branches occur regardless of the contents of the BESCR. See Section 9.4.4 of Book III and Section 6.2.12 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.
### Chapter 7. Event-Based Branch Facility

#### 31 Performance Monitor Event-Based Exception Enable (PME)

- **0** Performance Monitor event-based exceptions are disabled.
- **1** Performance Monitor event-based exceptions are enabled until a Performance Monitor event-based exception occurs, at which time:
  - PME is set to 0
  - PMEO is set to 1

See Chapter 9 of Book III for information about Performance Monitor event-based exceptions and about the effects of this bit on the Performance Monitor.

---

#### 32:33 Transaction State (TS)

When an event-based branch occurs, hardware sets this field to indicate the transaction state of the processor when the event-based branch occurred.

The values and their associated meanings are as follows.

- **00** Non-transactional
- **01** Suspended
- **10** Transactional
- **11** Reserved

BESCR\(_{TS}\) is part of the Transactional Memory facility. (The entire BESCR is part of the Event-Based Branch facility.)

---

#### 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 9 of Book III for information about Performance Monitor event-based exceptions and about the effects of this bit on the Performance Monitor.

---

#### 34:62 Effective Address

062 63

**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.

---

#### 34:63 Event Status

**34:61** Reserved

**62** External Event-Based Exception Occurred (EEO)

- **0** An external EBB exception has not occurred since the last time software set this bit to 0.
- **1** An external EBB exception has occurred since the last time software set this bit to 0.

---

### 7.2.2 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.

<table>
<thead>
<tr>
<th>Effective Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>62 63</td>
</tr>
</tbody>
</table>

**Figure 12. Event-Based Branch Handler Register (EBBHR)**
7.2.3 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.

<table>
<thead>
<tr>
<th>Effective Address</th>
<th>//</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>62 63</td>
</tr>
</tbody>
</table>

**Figure 13. 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.
7.3 Event-Based Branch Instructions

Return from Event-Based Branch

XL-form

rfebb S

<table>
<thead>
<tr>
<th>19</th>
<th>16</th>
<th>11</th>
<th>8</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>19</td>
<td>16</td>
<td>11</td>
<td>8</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

BESCRGE ← S
MSR_TS ← BESCRTS
NIA ← lea EBBRR0:61 || 0b00

BESCRGE is set to S. The processor is placed in the transaction state indicated by BESCRTS.

If there are no pending event-based exceptions, then the next instruction is fetched from the address EBBRR0:61 || 0b00 (when MSR_SF=1) or EBBRR32:61 || 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 EBBRR 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.

See Section 3.4 of Book III for additional information about this instruction.

Special Registers Altered:

BESCR
MSR (See Book III)

Extended Mnemonics:

Extended: Equivalent to:
rfebb rfebb 1

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

If the BESCR_TS has been modified by software after an event-based branch occurs, an illegal transaction state transition may occur. See Chapter 3.2.2 of Book III.

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 pt
mspr 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 0000 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 0000 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 8. 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 7 of Book II.) When used in conjunction with this facility, BESCRPME 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 9 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 9.4.7 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 8.1.)

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 if allowed by the filtering mode; subsequently, bits 0:61 of the effective address of the branch target are written into the following entry.

BHRB entries are written as described above without regard to transaction state and are not removed due to transaction failures.

### 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.

BHRB entries are written as described above without regard to transaction state and are not removed due to transaction failures.
8.1 Branch History Rolling Buffer Entry Format

Branch History Rolling Buffer Entries (BHRBEs) have the following format.

<table>
<thead>
<tr>
<th>Effective Address</th>
<th>T</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>62</td>
<td>63</td>
</tr>
</tbody>
</table>

**Figure 14. Branch History Rolling Buffer Entry**

**0:61 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.

When the EA field contains a non-zero value, bits 62:63 have the following meanings.

**62 Target Address (T)**

0 The EA field contains bits 0:61 of the effective address of a Branch instruction for which the branch was taken.
1 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.

**63 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.

When the EA field contains a zero value, bits 62:63 specify the type of marker as described below.

<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>

**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.
8.2 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 9 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

\[ \text{clrbhrb} \]

for \( n = 0 \) to (number_of_BHRBEs implemented - 1)

\( \text{BHRB}(n) \leftarrow 0 \)

All BHRB entries are set to 0s.

Special Registers Altered:
None.

Move From Branch History Rolling Buffer Entry

\[ \text{mfbhrbe \ RT,BHRBE} \]

\[ \begin{array}{cccc|c}
31 & 30 & 29 & 28 & 31 \\
\hline
0 & 6 & 11 & 16 & 21 & 430 & 7 \\
\end{array} \]

\( n \leftarrow \text{BHRBE}_{0:9} \)

If \( n < \) number of BHRBEs implemented then
\( \text{RT} \leftarrow \text{BHRBE}(n) \)
else
\( \text{RT} \leftarrow 640 \)

The BHRBE field denotes an entry in the BHRB. If the designated entry is within the range of BHRB entries implemented and Performance Monitor aler ts are disable (see Section 9.5 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 clrbhrb instructions have completed.

- an event-based branch has occurred
- an rfebb (see Chapter 7 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 clrbhrb.
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.

\[ \text{dcbtct RA, RB, TH} \] (equivalent to: \text{dcbt for TH values of 0b00000 - 0b00111}; other TH values are invalid.

\[ \text{dcbtds RA, RB, TH} \] (equivalent to: \text{dcbt for TH values of 0b00000 or 0b01000 - 0b01111}; other TH values are invalid.

\[ \text{dcbtt RA, RB} \] (equivalent to: \text{dcbt for TH value of 0b10000})

\[ \text{dbbna RA, RB} \] (equivalent to: \text{dcbt for TH value of 0b10001})

\[ \text{dcbtstct RA, RB, TH} \] (equivalent to: \text{dcbtst for TH values of 0b00000 or 0b00000 - 0b01111}; other TH values are invalid.

\[ \text{dcbtstds RA, RB, TH} \] (equivalent to: \text{dcbtst for TH values of 0b00000 or 0b01000 - 0b01111}; other TH values are invalid.

\[ \text{dcbtstt RA, RB} \] (equivalent to: \text{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. Extended mnemonics are provided that represent the L value in the mnemonic rather than requiring it to be coded as a numeric operand.

\[ \text{dcbf RA, RB} \] (equivalent to: \text{dcbf RA, RB, 0})

\[ \text{dcbfl RA, RB} \] (equivalent to: \text{dcbf RA, RB, 1})

\[ \text{dcbfip RA, RB} \] (equivalent to: \text{dcbf RA, RB, 3})

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.

\[ \text{miso} \] (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.

\[ \text{Note: lbarx, lhax, lwax, ldax, and lqax} \] 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.

lbarx RT,RA,RB (equivalent to: lbarx RT,RA,RB,0)
lharx RT,RA,RB (equivalent to: lharcx RT,RA,RB,0)
lwarx RT,RA,RB (equivalent to: lwarcx RT,RA,RB,0)
ldarx RT,RA,RB (equivalent to: ldarcx RT,RA,RB,0)
lqarcx RT,RA,RB (equivalent to: lcqarcx RT,RA,RB,0)

A.5 Synchronize Mnemonics

The L field in the Synchronize instruction controls the scope of the synchronization function performed by the instruction. Extended mnemonics are provided that represent the L value in the mnemonic rather than requiring it to be coded as a numeric operand. Two extended mnemonics are provided for the L=0 value in order to support Assemblers that do not recognize the sync mnemonic.

Note: sync serves as both a basic and an extended mnemonic. Assemblers will recognize a sync mnemonic with one operand as the basic form, and a sync mnemonic with no operand as the extended form. In the extended form the L operand is omitted and assumed to be 0.

sync (equivalent to: sync 0)
lwsync (equivalent to: sync 1)
ptesync (equivalent to: sync 2)

A.6 Wait Mnemonics

The WC field in the wait instruction is reserved for future use. It may be be used in the future to indicate the condition that causes instruction execution to resume. An extended mnemonic is provided that represents the WC value in the mnemonic rather than requiring it to be coded as a numeric operand.

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

wait (equivalent to: wait 0)

tend. (equivalent to: tend. 0)
tendall. (equivalent to: tend. 1)

dsuspend. (equivalent to: tsk. 0)
tresume. (equivalent to: tsk. 1)

A.8 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 (equivalent to: mftb Rx,268)
or: mfspr Rx,268
mftbu Rx (equivalent to: mftb Rx,269)
or: mfspr Rx,269

A.9 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 (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:
    lwarx 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 lwarx. While the store is redundant with respect to the value in the location, its success ensures that the value loaded by the lwarx 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:
    lwarx 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
```

**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 #if not
    stwcx. r5,0,r3 #store new value if still res’ved
    bne- loop #loop if lost reservation
exit: ...
```

**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 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 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 lwarx and stwcx. instructions are used to obtain the lock, an import barrier can be constructed by placing an isync instruction immediately following the loop containing the lwarx and 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, the value to which the lock should be set is in GPR 5, the old value of the lock is returned in GPR 6, and the address of the shared data structure is in GPR 9.

```
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 lwarx indicates that after the program acquires the lock variable (i.e., stwcx. is successful), it will release it (i.e., store to it) prior to another program attempting to modify it.

The second bne- does not complete until CR0 has been set by the stwcx.. The stwcx. does not set CR0 until it has completed (successfully or unsuccessfully). The lock is acquired when the stwcx. completes successfully. Together, the second bne- and the subsequent isync create an import barrier that prevents the load from “data1” from being performed until the branch has been resolved not to be taken.

If the shared data structure is in storage that is neither Write Through Required nor Caching Inhibited, an lwsync instruction can be used instead of the isync instruction. If lwsync is used, the load from “data1” may be performed before the stwcx.. But if the stwcx. fails, the second branch is taken and the lwarx is re-executed. If the stwcx. succeeds, the value returned by the load from “data1” is valid even if the load is performed before the stwcx., because the lwsync ensures that the load is performed after the instance of the lwarx that created the reservation used by the successful stwcx..

B.2.1.2 Obtain Pointer and Import Shared Storage

If lwarx and 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.

```
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 lwarx. The load from “data1” may be performed before the stwcx.. But if the stwcx. fails, the branch is taken and the value returned by the load from “data1” is discarded. If the stwcx. succeeds, the value returned by the load from “data1” is valid even if the load is performed before the stwcx., because the load uses the pointer value returned by the instance of the lwarx that created the reservation used by the successful stwcx..

An isync instruction could be placed between the bne- and the subsequent lwz, but no isync is needed if all accesses to the shared data structure depend on the value returned by the lwarx.
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 9.

    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.
B.3 List Insertion

This section shows how the `lwarx` and `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.

```assembly
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.

```assembly
loop1:
    lwz r2,0(r3)#load the word
    cmpwi r5,0 #loop back if word
    bne- loop #  not equal to 0
    lwarx r5,0,r3 #try to store non-0
    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 1.7.4.1) between the `lwarx` and the `stwcx` of a `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.

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 `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 `stwcx`. instructions might be executed, and reservations might more often be lost between the `lwarx` and the `stwcx`.

2. The manner in which `lwarx` and `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 `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.

```assembly
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 1.7.4.1) between the `lwarx` and the `stwcx` of a `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.

B.5 Transactional Lock Elision

This section illustrates the use of the Transactional Memory facility to implement transactional lock elision (TLE), in which lock-based critical sections are speculatively executed as a transaction without first acquiring a lock. This locking protocol is an alternative to the routines described above, yielding increased concurrency when the lock that guards a critical section is frequently unnecessary.
B.5.1 Enter Critical Section

The following example shows the entry point to a critical section using transactional lock elision. The entry code starts a transaction using the `tbegin` instruction and checks whether the transaction was aborted or not. If not, it checks whether the lock is free or not. If the lock is found to be free, the thread proceeds to execute the critical section.

In this example it is assumed that the address of the lock is in GPR 3, and the value indicating that the lock is free is in GPR 4. The handling of cases of transaction abort and busy lock are described in subsequent examples.

```assembly
tle_entry:
tbegin.             #Start TLE transaction
beq- tle_abort     #Handle TLE transaction abort
lwz r6,0(r3)       #Read lock
cmpw r6,r4         #Check if lock is free
bne- busy_lock     #If not, handle lock busy case

critical_section1:
```

B.5.2 Handling Busy Lock

In the event that the lock is already held, by either another thread or the current thread, the transaction is aborted using the `tabort` instruction, using a software-defined code `TLE_BUSY_LOCK` indicating the cause of the abort. The abort returns control to the `beq` following `tbegin`, in the critical section entrance sequence, allowing for an abort handler to react appropriately.

```assembly
busy_lock:
        li r3, TLE_BUSY_LOCK
        tabort r3          #Abort TLE transaction
```

B.5.3 Handling TLE Abort

A TLE transaction may fail for one of a variety of causes, persistent and transient. Persistent causes are certain—or at least highly likely—to cause future attempts to execute the same transaction to fail. However, for transient causes, it is possible that the failure cause may not be re-encountered in a subsequent attempt. Thus, persistent aborts are handled by taking a non-transactional path that involves the actual acquisition of the lock, while transient aborts retry the critical section using TLE.

The following example illustrates the handling of aborts in TLE. It is assumed that the address of the lock is in GPR 3. The immediate value of the `andis` instruction selects the Failure Persistent bit in the upper half of TEXASR to be tested.

```assembly
tle_abort:
        mf spr r4, TEXASRU # Read high-order half
        # of TEXASR
        andis. r5, r4, 0x0100 # determine whether failure
        # is likely to be persistent
        bne tle_acquire_lock #Persistent, acquire lock
        #enter critical sec
        b tle_entry          #Transient, try TLE again
```

This example can be extended to keep track of the number of transient aborts and fall back on the acquisition of the lock after the number of transient failures reaches some threshold. It can also be extended to handle reentrant locks. Acquisition of TLE locks is described in a subsequent example.

B.5.4 TLE Exit Section Critical Path

The following example illustrates the instruction sequence used to exit a TLE critical section. The CR0 value set by `tend` indicates whether the current thread was in a transaction. If so, the exited critical section was entered speculatively, and the transaction is ended. If not, the execution takes a path to release the lock.

Release of an acquired TLE lock is described in a subsequent example.

```assembly
tle_exit:
        tend.             #End the current trans-
        #action, if any
        bng- tle_release_lock #Release lock, if was
        #not in a transaction
```

B.5.5 Acquisition and Release of TLE Locks

The steps for acquiring and releasing a lock associated with a TLE critical section are identical to those for acquiring and releasing conventional locks that are not elided, as described in Section B.2.1.1 and Section B.2.2 respectively.

Programming Note

A future version of the architecture will revise the `isync` and `lwsync` instruction descriptions to make them consistent with the use of these instructions, as shown in Section B.2.1.1, to acquire a lock associated with a TLE critical section.
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 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 lookaside 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.

- **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 6. "Interrupts" on page 1049.

- **trap interrupt**
  An interrupt that results from execution of a Trap instruction.

- **Additional exceptions**
  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 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 Machine Check interrupt. (The determination is affected by the potential occurrence of any other kind of interrupt.)
  - A context-altering instruction is executed (Chapter 11. "Synchronization Requirements for Context Alterations" on page 1133). 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 8.2 of Book III) need not occur until a subsequent context synchronizing operation has occurred.
  - An mtgsr is executed and an interrupt occurs before the mtspr sequence following mtgsr has finished executing. The contents of SPRs that are the targets of mtspRs instructions between the point of interruption and the end of the mtspRs sequence may be altered.

- **“must”**
  If hypervisor software 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.

- **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.

- **hypervisor privileged**
  A term used to describe an instruction or facility that is available only when the thread is in hypervisor 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.

- **I, II, III, ...**
  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
Chapter 1. Introduction

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 MSRFP=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., \texttt{tlbie}) when the thread is in privileged but non-hypervisor 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 \textit{context synchronizing} if it satisfies the requirements listed below. Such instructions and events are collectively called \textit{context synchronizing operations}. The context synchronizing operations are the \texttt{isync} instruction, the \texttt{System Linkage} instructions, the \texttt{mtmsr[d]} instructions with \texttt{L}=0, and most interrupts (see Section 6.4).

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 \texttt{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., \texttt{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 6.9).

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 5.5, “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 sync and ptesync; see Section 1.5.2) because all other execution synchronizing operations also alter context.

1.5.2 Execution Synchronization

An instruction is *execution synchronizing* if it satisfies items 2 and 3 of the definition of context synchronization (see Section 1.5.1). sync and ptesync are treated like isync with respect to item 2. The execution synchronizing instructions are sync, ptesync, the mtmsr[d] instructions with L=1, and all context synchronizing instructions.

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.
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 sub-sections, 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_HPRI_PR=0b10) are performed as if VC=0b0000.</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 5.7.2, “Virtualized Partition Memory (VPM) Mode”, and Section 5.7.3.3, “Virtual Real Mode Addressing Mechanism”, 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).

2 Ignore SLB Large Page Specification (ISL)

Controls whether ISL mode is enabled as specified below.

0 - ISL mode disabled
1 - ISL mode enabled

When ISL mode is enabled and address translation is enabled, address translation is performed as if the contents of SLB\_LILP and PRTE\_STRPS were 0b000. When address translation is disabled, the setting of the ISL
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_HV[IR]=0b10, Virtual Page Class Key Storage Protection exceptions that occur on storage operand accesses when VPM=0 cause Hypervisor Data Storage interrupts.

38 Interrupt Little-Endian (ILE)

The contents of the ILE bit are copied into MSR_LE by interrupts that set MSRHV to 0 (see Section 6.5), to establish the Endian mode for the interrupt handler.

39:40 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 except those subject to the overrides described below.

0 The interrupt is taken with MSR_IR_DR = 0b00 and no effective address offset or alternate effective address.
1 Reserved
2 The interrupt is taken with MSR_IR_DR = 0b11. If the interrupt is not System Call Vectored, an effective address offset of 0x0000_0000_0000_8000 is applied. System Call Vectored does not use an alternate effective address.
3 The interrupt is taken with MSR_IR_DR = 0b11. If the interrupt is not System Call Vectored, an effective address offset of 0x0000_0000_0000_4000 is applied. System Call Vectored uses an alternate effective address of 0xc000_0000_0000_3 || LEV || 0b_0000.

Machine Check, System Reset, and Hypervisor Maintenance interrupts are taken as if LPCRAIL=0. In the remainder of this definition, “other interrupts” means interrupts other than these three.

Other interrupts that occur when MSR_IR=0 or MSR_DR=0, are taken as if LPCR_AIL=0.

When the hypervisor receiving the other interrupts uses HPT translation and the interrupts have caused a transition from MSRHV=0 to
One of the purposes of the AIL field is to provide relocation for interrupts that occur while an application is running with MSR$_{HV}$ PR=0b11 under a "bare metal" operating system (i.e., an operating system that runs in hypervisor state), such as KVM.

**Programming Note**

Running with LPCR$_{EVIRT}$=1 facilitates support of nested hypervisors (hypervisors that run with MSR$_{HV}$ PR=0b00 and have their use of hypervisor resources virtualized by a higher level hypervisor); see the relevant Programming Note in Section 6.5.18, “Hypervisor Emulation Assistance Interrupt”. It also permits emulation of new SPRs on designs that do not support them in hardware.

**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$_{UPRT}$=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 access hypervisor resources or execute hypervisor privileged instructions in privileged but non-hypervisor state cause a Privileged Instruction type Program interrupt; attempts to access undefined SPR numbers (using mtspr or mfspr) other than 0, 4, 5, and 6 in privileged state are treated as no-ops.
- 1 Enhanced Virtualization is enabled: attempts to access hypervisor resources or execute hypervisor privileged instructions in privileged but non-hypervisor state cause a Hypervisor Emulation Assistance interrupt; attempts to access undefined SPR numbers (using mtspr or mfspr) other than 0, 4, 5, and 6 in privileged state cause a Hypervisor Emulation Assistance interrupt.

**Programming Note**

Running with LPCR$_{EVIRT}$=1 facilitates support of nested hypervisors (hypervisors that run with MSR$_{HV}$ PR=0b00 and have their use of hypervisor resources virtualized by a higher level hypervisor); see the relevant Programming Note in Section 6.5.18, “Hypervisor Emulation Assistance Interrupt”. It also permits emulation of new SPRs on designs that do not support them in hardware.

**Host Radix (HR)**

Indicates whether the partition uses Radix Tree translation, as specified below.

- 0 Hypervisor does not use Radix Tree translation.
- 1 Hypervisor uses Radix Tree translation.

**Programming Note**

The hypervisor must program HR to match the Host Radix bit in the appropriate Partition Table Entry. If the values do not match, the results are undefined.

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.

**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.

**Large Decrementer (LD)**

0 Large Decrementer mode is not enabled.
1 Large Decrementer mode is enabled.

See Section 7.4 for additional information.
Privileged Doorbell Exit Enable

0 When the **stop** instruction is executed with PSSC\_\_EC=1, Directed Privileged Doorbell exceptions are not enabled to cause exit from power-saving mode.

1 When the **stop** instruction is executed with PSSC\_\_EC=1, Directed Privileged Doorbell exceptions are enabled to cause exit from power-saving mode.

Hypervisor Doorbell Exit Enable

0 When the **stop** instruction is executed with PSSC\_\_EC=1, Directed Hypervisor Doorbell exceptions are not enabled to cause exit from power-saving mode.

1 When the **stop** instruction is executed with PSSC\_\_EC=1, Directed Hypervisor Doorbell exceptions are enabled to cause exit from power-saving mode.

External Exit Enable

0 When the **stop** instruction is executed with PSSC\_\_EC=1, External exceptions are not enabled to cause exit from power-saving mode.

1 When the **stop** instruction is executed with PSSC\_\_EC=1, External exceptions are enabled to cause exit from power-saving mode.

Decrementer Exit Enable

0 When the **stop** instruction is executed with PSSC\_\_EC=1, Decrementer exceptions are not enabled to cause exit from power-saving mode.

1 When the **stop** instruction is executed with PSSC\_\_EC=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.)

Other Exit Enable

0 When the **stop** instruction is executed with PSSC\_\_EC=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 PSSC\_\_EC=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 6.5.1 and Section 6.5.2 for additional information about exit from power-saving mode.

Mediated External Exception Request (MER)

0 A Mediated External exception is not requested.

1 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.

- LPC\_MER = 1 and a Mediated External exception exists.
- LPC\_MER = 0 and a Mediated External exception does not exist.

A context synchronizing instruction or event that is executed or occurs when LPC\_MER = 0 ensures that the exception effects of LPC\_MER are consistent with the contents of LPC\_MER. Otherwise, when an instruction changes the contents of LPC\_MER, the exception effects of LPC\_MER become consistent with the new contents of LPC\_MER reasonably soon after the change.

Programming Note

LPC\_MER 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 LPC\_MER, the partition can be interrupted upon enabling external interrupts. Without using LPC\_MER, 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**, **slibeg**, and **slbiag** directly, or must issue a system call to the hypervisor.

0 Guest is not permitted to use **tlbie**, **slibeg**, **slbiag**, **tlbsync**, and **slbsync**.

1 Guest is permitted to use **tlbie**, **slibeg**, **slbiag**, **tlbsync**, and **slbsync**.
Translation Control (TC)

0 The secondary Page Table search is enabled.
1 The secondary Page Table search is disabled.

55:58 Reserved

Hypervisor External Interrupt Control (HEIC)

0 Direct External interrupts can occur in Hypervisor state.
1 Direct External interrupts cannot occur in hypervisor state.

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.

Hypervisor Virtualization Interrupt Conditionally Enable (HVICE)

0 Hypervisor Virtualization interrupts are disabled.
1 Hypervisor Virtualization interrupts are enabled if permitted by MSR$_{EE}$, MSR$_{HV}$, and MSR$_{PR}$; see Section 6.5.21.

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$_{PR}$; see Section 6.5.12 on page 1077.

See Section 6.5 on page 1063 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](image)

<table>
<thead>
<tr>
<th>Bits</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-4</td>
<td>HRMO</td>
<td>Real Mode Offset</td>
</tr>
</tbody>
</table>

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 5.7.3 on page 984 and Section 5.7.5 on page 987.

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](image)

<table>
<thead>
<tr>
<th>Bits</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>32:63</td>
<td>LPID</td>
<td>Logical Partition Identifier</td>
</tr>
</tbody>
</table>

The contents of the LPIDR identify the partition to which the thread is assigned, affecting some aspects of translation and interrupt delivery. 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](image)

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. Facilities that are made unavailable by the PCR are treated as follows when the thread is in problem state:

- Instructions are treated as illegal instructions.
- 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.

- Unless the second item of this list applies, bits in system registers read back 0s for mspr 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. mfxer 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 XEROV or XERCA, 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.

The txer, seli[], selr[], and selrr[] instructions read bits 44:45 of the XER as 0s, independent of the PCR.

- 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 defined bit in the PCR may also control whether certain instructions, SPRs, and other related facilities are available in a privileged state (MSRPR=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 PC bit may also determine how an instruction field value is interpreted or may define other behavior as specified in the bit definitions below.

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.
[h]rfid and mtmsr[d] instructions, except as specified elsewhere in this section.

When facilities that have enable bits in the MSR, FSCR, HFSCR, or MMCR0 are made unavailable by the value in the PCR, they become unavailable in problem state as specified above regardless of whether they are enabled by the corresponding MSR, FSCR, HFSCR, or MMCR0 bit; facility availability interrupts (e.g. [Hypervisor] Facility Available, Vector Unavailable, etc.) do not occur as a result of problem state accesses even if the corresponding field in the MSR, [H]FSCR, or MMCR0 makes them unavailable in problem state.

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 Transactional Memory, the BHBRB 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.

The bit definitions for the PCR are shown below.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:59</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

The instructions listed in Table 1 do not occur as a result of problem state accesses even if the corresponding field in the MSR, [H]FSCR, or MMCR0 makes them unavailable in problem state.

Table 1: Instructions Controlled by the V2.07 Bit

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</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 Qword</td>
</tr>
<tr>
<td>bcdcfz.</td>
<td>Decimal Convert From Zoned</td>
</tr>
<tr>
<td>bcdcpsgn</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 Qword</td>
</tr>
<tr>
<td>bcdctz.</td>
<td>Decimal Convert To Zoned</td>
</tr>
<tr>
<td>bcds.</td>
<td>Decimal Shift</td>
</tr>
<tr>
<td>bcdsetsgn.</td>
<td>Decimal Set Sign</td>
</tr>
<tr>
<td>bcdsr.</td>
<td>Decimal Shift and Round</td>
</tr>
<tr>
<td>bcdtrunc.</td>
<td>Decimal Truncate</td>
</tr>
<tr>
<td>bcdus.</td>
<td>Decimal Unsigned Shift</td>
</tr>
</tbody>
</table>

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 the version of the architecture subsequent to Version 2.07.

- The instructions listed in Table 1
- scv
- The splitting out of footprint overflows in which other threads contributed to the problem to set TEXASR_{17} and indicate a transient failure instead of setting TEXASR_{10} and indicating a persistent failure.

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.

When MSR{\textsubscript{PR}}=0 (i.e., privileged or hypervisor-privileged state), this bit controls the availability of the mcrrxx instruction and which bit pair is read by mfxer for XER bits 44:45.

0 mcrrxx is available.

- mfxer reads the contents of the “OV32-CA32” bit pair for XER bits 44:45.

1 mcrrxx is unavailable.

- mfxer reads the contents of the “reserved” bit pair for XER bits 44:45.
<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>bcdutrcn</td>
<td>Decimal Unsigned Truncate</td>
</tr>
<tr>
<td>cmpeqb</td>
<td>Compare Equal Byte</td>
</tr>
<tr>
<td>cmprb</td>
<td>Compare Ranged Byte</td>
</tr>
<tr>
<td>cnttzdw[]</td>
<td>Count Trailing Zeros Dword</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>darn</td>
<td>Deliver a Random Number</td>
</tr>
<tr>
<td>dtstfsi</td>
<td>DFP Test Significance Immediate</td>
</tr>
<tr>
<td>dtstfsiq</td>
<td>DFP Test Significance Immediate Quad</td>
</tr>
<tr>
<td>extswsl[i][]</td>
<td>Extend Sign Word and Shift Left Immediate</td>
</tr>
<tr>
<td>ldat</td>
<td>Load Doubleword Atomic</td>
</tr>
<tr>
<td>lwat</td>
<td>Load Word Atomic</td>
</tr>
<tr>
<td>lxsd</td>
<td>Load VSX Scalar Dword</td>
</tr>
<tr>
<td>lxsbzx</td>
<td>Load VSX Scalar as Integer Byte &amp; Zero Indexed</td>
</tr>
<tr>
<td>lxsihx</td>
<td>Load VSX Scalar as Integer Hword &amp; Zero Indexed</td>
</tr>
<tr>
<td>lxssp</td>
<td>Load VSX Scalar Single</td>
</tr>
<tr>
<td>lxv</td>
<td>Load VSX Vector</td>
</tr>
<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>lxvwxs</td>
<td>Load VSX Vector Word &amp; Splat Indexed</td>
</tr>
<tr>
<td>lxvx</td>
<td>Load VSX Vector Indexed</td>
</tr>
<tr>
<td>maddhd</td>
<td>Multiply-Add High Dword</td>
</tr>
<tr>
<td>maddhdu</td>
<td>Multiply-Add High Dword Unsigned</td>
</tr>
<tr>
<td>maddld</td>
<td>Multiply-Add Low Dword</td>
</tr>
<tr>
<td>mcrxx</td>
<td>Move XER to CR Extended</td>
</tr>
<tr>
<td>mffsce</td>
<td>Move From FPSCR &amp; Clear Enables</td>
</tr>
<tr>
<td>mffscdrl</td>
<td>Move From FPSCR Control &amp; set DRN</td>
</tr>
<tr>
<td>mffscdri</td>
<td>Move From FPSCR Control &amp; set DRN Immediate</td>
</tr>
<tr>
<td>mffscren</td>
<td>Move From FPSCR Control &amp; set RN</td>
</tr>
<tr>
<td>mffscreni</td>
<td>Move From FPSCR Control &amp; set RN Immediate</td>
</tr>
<tr>
<td>mffsl</td>
<td>Move From FPSCR Lightweight</td>
</tr>
<tr>
<td>mfvsrld</td>
<td>Move From VSR Lower Dword</td>
</tr>
<tr>
<td>modsd</td>
<td>Modulo Signed Dword</td>
</tr>
<tr>
<td>modsw</td>
<td>Modulo Signed Word</td>
</tr>
<tr>
<td>modud</td>
<td>Modulo Unsigned Dword</td>
</tr>
<tr>
<td>moduw</td>
<td>Modulo Unsigned Word</td>
</tr>
<tr>
<td>mtvsvrd</td>
<td>Move To VSR Double Dword</td>
</tr>
<tr>
<td>mtvsws</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 Dword</td>
</tr>
<tr>
<td>stxsibx</td>
<td>Store VSX Scalar as Integer Byte Indexed</td>
</tr>
<tr>
<td>stxsibx</td>
<td>Store VSX Scalar as Integer Hword Indexed</td>
</tr>
</tbody>
</table>

Table 1: Instructions Controlled by the V 2.07 Bit
<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>stxssp</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>stxvh8x</td>
<td>Store VSX Vector Halfword*8 Indexed</td>
</tr>
<tr>
<td>stxvl</td>
<td>Store VSX Vector with Length</td>
</tr>
<tr>
<td>stxvl</td>
<td>Store VSX Vector Left-justified with Length</td>
</tr>
<tr>
<td>stxvx</td>
<td>Store VSX Vector Indexed</td>
</tr>
<tr>
<td>vabsdub</td>
<td>Vector Absolute Difference Unsigned Byte</td>
</tr>
<tr>
<td>vabsduh</td>
<td>Vector Absolute Difference Unsigned Hword</td>
</tr>
<tr>
<td>vabsduw</td>
<td>Vector Absolute Difference Unsigned Word</td>
</tr>
<tr>
<td>vbpermd</td>
<td>Vector Bit Permute Dword</td>
</tr>
<tr>
<td>vclzlsbb</td>
<td>Vector Count Leading Zero Least-Significant Bits Byte</td>
</tr>
<tr>
<td>vcmpneb[]</td>
<td>Vector Compare Not Equal Byte</td>
</tr>
<tr>
<td>vcmpneh[]</td>
<td>Vector Compare Not Equal Hword</td>
</tr>
<tr>
<td>vcmpnew[]</td>
<td>Vector Compare Not Equal Word</td>
</tr>
<tr>
<td>vcmpnezb[]</td>
<td>Vector Compare Not Equal or Zero Byte</td>
</tr>
<tr>
<td>vcmpnezh[]</td>
<td>Vector Compare Not Equal or Zero Hword</td>
</tr>
<tr>
<td>vcmpnezw[]</td>
<td>Vector Compare Not Equal or Zero Word</td>
</tr>
<tr>
<td>vctzb</td>
<td>Vector Count Trailing Zeros Byte</td>
</tr>
<tr>
<td>vctzd</td>
<td>Vector Count Trailing Zeros Dword</td>
</tr>
<tr>
<td>vctzh</td>
<td>Vector Count Trailing Zeros Hword</td>
</tr>
<tr>
<td>vclzlsbb</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 Dword</td>
</tr>
<tr>
<td>vextractub</td>
<td>Vector Extract Unsigned Byte</td>
</tr>
<tr>
<td>vextractuh</td>
<td>Vector Extract Unsigned Hword</td>
</tr>
<tr>
<td>vextractuw</td>
<td>Vector Extract Unsigned Word</td>
</tr>
<tr>
<td>vextsb2d</td>
<td>Vector Extend Sign Byte To Dword</td>
</tr>
<tr>
<td>vextsb2w</td>
<td>Vector Extend Sign Byte To Word</td>
</tr>
<tr>
<td>vextsh2d</td>
<td>Vector Extend Sign Hword To Dword</td>
</tr>
<tr>
<td>vextsh2w</td>
<td>Vector Extend Sign Hword To Word</td>
</tr>
<tr>
<td>vextsw2d</td>
<td>Vector Extend Sign Word To Dword</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 Hword Left-Indexed</td>
</tr>
<tr>
<td>vextuhrx</td>
<td>Vector Extract Unsigned Hword Right-Indexed</td>
</tr>
<tr>
<td>vextuwlx</td>
<td>Vector Extract Unsigned Word Left-Indexed</td>
</tr>
<tr>
<td>vextuwxr</td>
<td>Vector Extract Unsigned Word Right-Indexed</td>
</tr>
<tr>
<td>vinsertb</td>
<td>Vector Insert Byte</td>
</tr>
<tr>
<td>vinsertd</td>
<td>Vector Insert Dword</td>
</tr>
<tr>
<td>vinserth</td>
<td>Vector Insert Hword</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 Qword</td>
</tr>
<tr>
<td>vmul10ecuq</td>
<td>Vector Multiply-by-10 Extended &amp; write Carry Unsigned Qword</td>
</tr>
<tr>
<td>vmul10eucq</td>
<td>Vector Multiply-by-10 Extended Unsigned Qword</td>
</tr>
<tr>
<td>vmul10ueq</td>
<td>Vector Multiply-by-10 Unsigned Qword</td>
</tr>
<tr>
<td>vnegd</td>
<td>Vector Negate Dword</td>
</tr>
<tr>
<td>vnegw</td>
<td>Vector Negate Word</td>
</tr>
</tbody>
</table>

Table 1: Instructions Controlled by the V 2.07 Bit
<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>vpermr</td>
<td>Vector Permute Right-indexed</td>
</tr>
<tr>
<td>vprtybd</td>
<td>Vector Parity Byte Dword</td>
</tr>
<tr>
<td>vprtybq</td>
<td>Vector Parity Byte Qword</td>
</tr>
<tr>
<td>vprtybw</td>
<td>Vector Parity Byte Word</td>
</tr>
<tr>
<td>vridmi</td>
<td>Vector Rotate Left Dword then Mask Insert</td>
</tr>
<tr>
<td>vrdnm</td>
<td>Vector Rotate Left Dword then AND with Mask</td>
</tr>
<tr>
<td>vrlwmi</td>
<td>Vector Rotate Left Word then Mask Insert</td>
</tr>
<tr>
<td>vrlwmm</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[o]</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>xscmpexpqqp</td>
<td>VSX Scalar Quad-Precision Compare Exponents</td>
</tr>
<tr>
<td>xscmpoqpp</td>
<td>VSX Scalar Quad-Precision Compare Ordered</td>
</tr>
<tr>
<td>xscmpuqpp</td>
<td>VSX Scalar Quad-Precision Compare Unordered</td>
</tr>
<tr>
<td>xspsgnqp</td>
<td>VSX Scalar Quad-Precision CopySign</td>
</tr>
<tr>
<td>xscvdqpp</td>
<td>VSX Scalar Quad-Precision Convert From Double-Precision</td>
</tr>
<tr>
<td>xscvhpqp</td>
<td>VSX Scalar Convert Half-Precision to Double-Precision</td>
</tr>
<tr>
<td>xscvqdpdq[o]</td>
<td>VSX Scalar round &amp; Convert Quad-Precision to Double-Precision [using round to Odd]</td>
</tr>
<tr>
<td>xscvqspsdz</td>
<td>VSX Scalar truncate &amp; Convert Quad-Precision to Signed Dword</td>
</tr>
<tr>
<td>xscvqpswz</td>
<td>VSX Scalar truncate &amp; Convert Quad-Precision to Signed Word</td>
</tr>
<tr>
<td>xscvqpsudz</td>
<td>VSX Scalar truncate &amp; Convert Quad-Precision to Unsigned Dword</td>
</tr>
<tr>
<td>xscvqpuwz</td>
<td>VSX Scalar truncate &amp; Convert Quad-Precision to Unsigned Word</td>
</tr>
<tr>
<td>xscvsdqpp</td>
<td>VSX Scalar Convert Signed Dword format to Quad-Precision format</td>
</tr>
<tr>
<td>xscvsphp</td>
<td>VSX Scalar round &amp; Convert Double-Precision to Half-Precision</td>
</tr>
<tr>
<td>xscvudqpp</td>
<td>VSX Scalar Convert Unsigned Dword format to Quad-Precision format</td>
</tr>
<tr>
<td>xsdivqpp[o]</td>
<td>VSX Scalar Quad-Precision Divide [&amp; round to Odd]</td>
</tr>
<tr>
<td>xsieexpdp</td>
<td>VSX Scalar Double-Precision Insert Exponent</td>
</tr>
<tr>
<td>xsieexpqpp</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>xsmmsubqp[o]</td>
<td>VSX Scalar Quad-Precision Multiply-Subtract [&amp; round to Odd]</td>
</tr>
<tr>
<td>xsmulqp[o]</td>
<td>VSX Scalar Quad-Precision Multiply [&amp; round to Odd]</td>
</tr>
<tr>
<td>xsabsqp</td>
<td>VSX Scalar Quad-Precision Negative Absolute</td>
</tr>
<tr>
<td>xsnegqp</td>
<td>VSX Scalar Quad-Precision Negate</td>
</tr>
<tr>
<td>xsnaddqp[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>xsrqpxp</td>
<td>VSX Scalar Quad-Precision Round to Double-Extended-Precision</td>
</tr>
<tr>
<td>xssqrtqpp[o]</td>
<td>VSX Scalar Quad-Precision Square Root [&amp; round to Odd]</td>
</tr>
<tr>
<td>xssubqpp[o]</td>
<td>VSX Scalar Quad-Precision Subtract [&amp; round to Odd]</td>
</tr>
<tr>
<td>xsttdcqp</td>
<td>VSX Scalar Double-Precision Test Data Class</td>
</tr>
<tr>
<td>xsttdcqp</td>
<td>VSX Scalar Quad-Precision Test Data Class</td>
</tr>
<tr>
<td>xsttdcsp</td>
<td>VSX Scalar Single-Precision Test Data Class</td>
</tr>
<tr>
<td>xsexexpqpp</td>
<td>VSX Scalar Double-Precision Extract Exponent</td>
</tr>
</tbody>
</table>

Table 1: Instructions Controlled by the V 2.07 Bit
This bit controls the availability, in problem state, of the following instructions, facilities, and behaviors that were newly available in problem state in the version of the architecture subsequent to Version 2.06.

- **icbt**
- **lq, stq lbarx, lhax, stbcx, sthcx**
- **lqarx, stqcx**
- **clrbrh, mfbhrbe**
- **rfebb, bctar[l]**
- The entire Transactional Memory facility
- The instructions in Table 2
- The reserved no-op instructions (see Section 1.9.3 of Book I)
- The reserved SPRs (see Section 1.3.3 of Book I)
- PPR32
- DSCR at SPR number 3
- SIER and MMCR2
- MMCR0:42:47, 51:55 and MMCRA0:63.

### Programming Note

The specified bits of MMCR0 and MMCRA above cannot be changed by **mtspr** instructions and **fmspr** instructions return 0s for these bits.

- **BESCR, EBBHR, and TAR**
- The ability of the or **31,31,31** and or **5,5,5** instructions to change the value of **PPRPR**.
- The ability of **mtspr** instructions that attempt to set **PPRPR** to 001 or 101 to change the value of **PPRPR**.

0 The instructions, facilities, and behaviors listed above are available in problem state.

1 The listed instructions, facilities, and behaviors listed above are unavailable in problem state.

If this bit is set to 1, then the V 2.07 bit must also be set to 1.
<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>bcdadd.</td>
<td>Decimal Add Modulo</td>
</tr>
<tr>
<td>bcdsub.</td>
<td>Decimal Subtract Modulo</td>
</tr>
<tr>
<td>fmrgew</td>
<td>Floating Merge Even Word</td>
</tr>
<tr>
<td>fmrgow</td>
<td>Floating Merge Odd Word</td>
</tr>
<tr>
<td>lxsiwax</td>
<td>Load VSX Scalar as Integer Word Algebraic Indexed</td>
</tr>
<tr>
<td>lxsiwzx</td>
<td>Load VSX Scalar as Integer Word and Zero Indexed</td>
</tr>
<tr>
<td>lxsisspx</td>
<td>Load VSX Scalar Single-Precision Indexed</td>
</tr>
<tr>
<td>mfvsrc</td>
<td>Move From VSR Doubleword</td>
</tr>
<tr>
<td>mfvsnwz</td>
<td>Move From VSR Word and Zero</td>
</tr>
<tr>
<td>mtvsrd</td>
<td>Move To VSR Doubleword</td>
</tr>
<tr>
<td>mtvsnwa</td>
<td>Move To VSR Word Algebraic</td>
</tr>
<tr>
<td>mtvsnzw</td>
<td>Move To VSR Word and Zero</td>
</tr>
<tr>
<td>stxsiwx</td>
<td>Store VSX Scalar as Integer Word Indexed</td>
</tr>
<tr>
<td>stxsspx</td>
<td>Store VSX Scalar Single-Precision Indexed</td>
</tr>
<tr>
<td>vaddcuq</td>
<td>Vector Add &amp; write Carry Unsigned Quadword</td>
</tr>
<tr>
<td>vaddecuq</td>
<td>Vector Add Extended &amp; write Carry Unsigned Quadword</td>
</tr>
<tr>
<td>vadduqm</td>
<td>Vector Add Extended Unsigned Quadword Modulo</td>
</tr>
<tr>
<td>vaddudm</td>
<td>Vector Add Unsigned Doubleword Modulo</td>
</tr>
<tr>
<td>vadduqm</td>
<td>Vector Add Unsigned Quadword Modulo</td>
</tr>
<tr>
<td>vbpermq</td>
<td>Vector Bit Permute Quadword</td>
</tr>
<tr>
<td>vcipher</td>
<td>Vector AES Cipher</td>
</tr>
<tr>
<td>vcipherlast</td>
<td>Vector AES Cipher Last</td>
</tr>
<tr>
<td>vclzb</td>
<td>Vector Count Leading Zeros Byte</td>
</tr>
<tr>
<td>vclzd</td>
<td>Vector Count Leading Zeros Doubleword</td>
</tr>
<tr>
<td>vclzfh</td>
<td>Vector Count Leading Zeros Halfword</td>
</tr>
<tr>
<td>vclzw</td>
<td>Vector Count Leading Zeros Word</td>
</tr>
<tr>
<td>vcmpequd[]</td>
<td>Vector Compare Equal To Unsigned Doubleword</td>
</tr>
<tr>
<td>vminpgtd[.]</td>
<td>Vector Compare Greater Than Signed Doubleword</td>
</tr>
<tr>
<td>vminpgtud[.]</td>
<td>Vector Compare Greater Than Unsigned Doubleword</td>
</tr>
<tr>
<td>veqv</td>
<td>Vector Logical Equivalence</td>
</tr>
<tr>
<td>vgbdb</td>
<td>Vector Gather Bits by Bytes by Doubleword</td>
</tr>
<tr>
<td>vmaxsd</td>
<td>Vector Maximum Signed Doubleword</td>
</tr>
<tr>
<td>vmaxud</td>
<td>Vector Maximum Unsigned Doubleword</td>
</tr>
<tr>
<td>vminsod</td>
<td>Vector Minimum Signed Doubleword</td>
</tr>
<tr>
<td>vminuod</td>
<td>Vector Minimum Unsigned Doubleword</td>
</tr>
<tr>
<td>vmrgew</td>
<td>Vector Merge Even Word</td>
</tr>
<tr>
<td>vmrgow</td>
<td>Vector Merge Odd Word</td>
</tr>
<tr>
<td>vmulesw</td>
<td>Vector Multiply Even Signed Word</td>
</tr>
<tr>
<td>vmuleuw</td>
<td>Vector Multiply Even Unsigned Word</td>
</tr>
<tr>
<td>vmoslsw</td>
<td>Vector Multiply Odd Signed Word</td>
</tr>
<tr>
<td>vmoslouw</td>
<td>Vector Multiply Odd Unsigned Word</td>
</tr>
<tr>
<td>vmuluwm</td>
<td>Vector Multiply Unsigned Word Modulo</td>
</tr>
<tr>
<td>vnand</td>
<td>Vector Logical NAND</td>
</tr>
</tbody>
</table>

Table 2: VSX and Vector Instructions Controlled by the v2.06 Bit
<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>vncipher</td>
<td>Vector AES Inverse Cipher</td>
</tr>
<tr>
<td>vncipherlast</td>
<td>Vector AES Inverse Cipher Last</td>
</tr>
<tr>
<td>vorc</td>
<td>Vector Logical OR with Complement</td>
</tr>
<tr>
<td>vpermxor</td>
<td>Vector Permute and Exclusive-OR</td>
</tr>
<tr>
<td>vpksdss</td>
<td>Vector Pack Signed Doubleword Signed Saturate</td>
</tr>
<tr>
<td>vpksdus</td>
<td>Vector Pack Signed Doubleword Unsigned Saturate</td>
</tr>
<tr>
<td>vpkdum</td>
<td>Vector Pack Unsigned Doubleword Unsigned Modulo</td>
</tr>
<tr>
<td>vpkdus</td>
<td>Vector Pack Unsigned Doubleword Unsigned Saturate</td>
</tr>
<tr>
<td>vpmsumb</td>
<td>Vector Polynomial Multiply-Sum Byte</td>
</tr>
<tr>
<td>vpmsumd</td>
<td>Vector Polynomial Multiply-Sum Doubleword</td>
</tr>
<tr>
<td>vpmsumh</td>
<td>Vector Polynomial Multiply-Sum Halfword</td>
</tr>
<tr>
<td>vpmsumw</td>
<td>Vector Polynomial Multiply-Sum Word</td>
</tr>
<tr>
<td>vpopcntb</td>
<td>Vector Population Count Byte</td>
</tr>
<tr>
<td>vpopcntd</td>
<td>Vector Population Count Doubleword</td>
</tr>
<tr>
<td>vpopcnth</td>
<td>Vector Population Count Halfword</td>
</tr>
<tr>
<td>vpopcntw</td>
<td>Vector Population Count Word</td>
</tr>
<tr>
<td>vrlf</td>
<td>Vector Rotate Left Doubleword</td>
</tr>
<tr>
<td>vsbox</td>
<td>Vector AES S-Box</td>
</tr>
<tr>
<td>vshasigmad</td>
<td>Vector SHA-512 Sigma Doubleword</td>
</tr>
<tr>
<td>vshasigmaw</td>
<td>Vector SHA-256 Sigma Word</td>
</tr>
<tr>
<td>vsl</td>
<td>Vector Shift Left Doubleword</td>
</tr>
<tr>
<td>vsrad</td>
<td>Vector Shift Right Algebraic Doubleword</td>
</tr>
<tr>
<td>vsrd</td>
<td>Vector Shift Right Doubleword</td>
</tr>
<tr>
<td>vsbucq</td>
<td>Vector Subtract &amp; write Carry Unsigned Quadword</td>
</tr>
<tr>
<td>vsbecuq</td>
<td>Vector Subtract Extended &amp; write Carry Unsigned Quadword</td>
</tr>
<tr>
<td>vsbeuqm</td>
<td>Vector Subtract Extended Unsigned Quadword Modulo</td>
</tr>
<tr>
<td>vsbudm</td>
<td>Vector Subtract Unsigned Doubleword Modulo</td>
</tr>
<tr>
<td>vsbuqwm</td>
<td>Vector Subtract Unsigned Quadword Modulo</td>
</tr>
<tr>
<td>vupkhsw</td>
<td>Vector Unpack High Signed Word</td>
</tr>
<tr>
<td>vupkisw</td>
<td>Vector Unpack Low Signed Word</td>
</tr>
<tr>
<td>xsaddsp</td>
<td>VSX Scalar Add Single-Precision</td>
</tr>
<tr>
<td>xscdpspn</td>
<td>Scalar Convert Double-Precision to Single-Precision format Non-signalling</td>
</tr>
<tr>
<td>xscdpspn</td>
<td>Scalar Convert Single-Precision to Double-Precision format Non-signalling</td>
</tr>
<tr>
<td>xscvsxdsp</td>
<td>VSX Scalar Convert Signed Fixed-Point Doubleword to Single-Precision</td>
</tr>
<tr>
<td>xscvsxdsp</td>
<td>VSX Scalar round and Convert Signed Fixed-Point Doubleword to Single-Precision format</td>
</tr>
<tr>
<td>xscvuxdsp</td>
<td>VSX Scalar Convert Unsigned Fixed-Point Doubleword to Single-Precision</td>
</tr>
<tr>
<td>xscvuxdsp</td>
<td>VSX Scalar round and Convert Unsigned Fixed-Point Doubleword to Single-Precision format</td>
</tr>
<tr>
<td>xsdivsp</td>
<td>VSX Scalar Divide Single-Precision</td>
</tr>
<tr>
<td>xsmdasp</td>
<td>VSX Scalar Multiply-Add Type-A Single-Precision</td>
</tr>
<tr>
<td>xsmdasp</td>
<td>VSX Scalar Multiply-Add Type-M Single-Precision</td>
</tr>
<tr>
<td>xsmsubasp</td>
<td>VSX Scalar Multiply-Subtract Type-A Single-Precision</td>
</tr>
<tr>
<td>xsmsubmsp</td>
<td>VSX Scalar Multiply-Subtract Type-M Single-Precision</td>
</tr>
<tr>
<td>xsmulisp</td>
<td>VSX Scalar Multiply Single-Precision</td>
</tr>
</tbody>
</table>

Table 2: VSX and Vector Instructions Controlled by the v2.06 Bit
This bit controls the availability, in problem state, of the following instructions, facilities, and behaviors that were newly available in problem state in the version of the architecture subsequent to Version 2.05.

- AMR access using SPR 13
- `addg6s`
- `bperm`
- `cdtdcd, cbcddt`
- `dcffix[]`
- `divde[0][], divdeu[0][], divwe[0][], divweu[0][]`
- `isel`
- `lfiwzx`
- `fctidu[], fctiduz[], fctiwu[], fctiwuz[], fcfids[], fcfidu[], fcfidus[], ftdiv, ftsqrt`
- `ldbrx, stdbrx`
- `popcntw, popcntd`
- All facilities in the VSX facility

0 The instructions, facilities, and behaviors listed above are available in problem state.

1 The instructions, facilities, and behaviors listed above are unavailable in problem state.

If this bit is set to 1, then the v2.06 bit must also be set to 1.

63 Reserved

The initial state of the PCR is all 0s.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>xsnmaddasp</td>
<td>VSX Scalar Negative Multiply-Add Type-A Single-Precision</td>
</tr>
<tr>
<td>xsnmaddmsp</td>
<td>VSX Scalar Negative Multiply-Add Type-M Single-Precision</td>
</tr>
<tr>
<td>xsnmsubasp</td>
<td>VSX Scalar Negative Multiply-Subtract Type-A Single-Precision</td>
</tr>
<tr>
<td>xsnmsubmsp</td>
<td>VSX Scalar Negative Multiply-Subtract Type-M Single-Precision</td>
</tr>
<tr>
<td>xsresp</td>
<td>VSX Scalar Reciprocal Estimate Single-Precision</td>
</tr>
<tr>
<td>xrsrpe</td>
<td>VSX Scalar Round to Single-Precision</td>
</tr>
<tr>
<td>xrsqrtresp</td>
<td>VSX Scalar Reciprocal Square Root Estimate Single-Precision</td>
</tr>
<tr>
<td>xssqrtsp</td>
<td>VSX Scalar Square Root Single-Precision</td>
</tr>
<tr>
<td>xssubsp</td>
<td>VSX Scalar Subtract Single-Precision</td>
</tr>
<tr>
<td>xleqv</td>
<td>VSX Logical Equivalence</td>
</tr>
<tr>
<td>xxlnand</td>
<td>VSX Logical NAND</td>
</tr>
<tr>
<td>xxlorc</td>
<td>VSX Logical OR with Complement</td>
</tr>
</tbody>
</table>

Table 2: VSX and Vector Instructions Controlled by the v2.06 Bit
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 4.4.4 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 4.4.4. (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.)

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.

2.7 Sharing Hypervisor 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 resources 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)
Threads that share any of the resources listed above, with the exception of the PTCR, the PVR and the HRMOR, must be in the same partition.

For each field of the LPCR, except the AIL, EVIR, 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.

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.

| 0 | 63 |

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 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 set MSRHV to 1 (see Section 6.5), 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. Branch Facility

3.1 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.

3.2 Branch Facility Registers

3.2.1 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 65 on page 1064. The MSR can also be modified by the mtmsr[d], rfscv, rfid, and hrfid instructions. It can be read by the mfmsr instruction.

![Figure 5. Machine State Register](image)

Below are shown the bit definitions for the Machine State Register.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
<th>MSR</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Sixty-Four-Bit Mode (SF)</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>The thread is in 32-bit mode.</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>The thread is in 64-bit mode.</td>
<td>1</td>
</tr>
<tr>
<td>1:2</td>
<td>Reserved</td>
<td>4, 5</td>
</tr>
<tr>
<td>3</td>
<td>Hypervisor State (HV)</td>
<td>6:28</td>
</tr>
<tr>
<td></td>
<td>The thread is not in hypervisor state.</td>
<td>29:30</td>
</tr>
<tr>
<td></td>
<td>If MSRPR=0 the thread is in hypervisor state; otherwise the thread is not in hypervisor state.</td>
<td></td>
</tr>
</tbody>
</table>

Programming Note

The privilege state of the thread is determined by MSR_{HV} and MSR_{PR}, as follows.

<table>
<thead>
<tr>
<th>HV</th>
<th>PR</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>privileged</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>problem</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>hypervisor</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>problem</td>
</tr>
</tbody>
</table>

Hypervisor state is also a privileged state (MSR_{PR} = 0). All references to "privileged state" in the Books include hypervisor state unless otherwise stated or obvious from context.

MSR_{HV} can be set to 1 only by the System Call instruction and some interrupts. It can be set to 0 only by rfid and hrfid.

It is possible to run an operating system in an environment that lacks a hypervisor, by always having MSR_{HV} = 1 and using MSR_{HV PR} = 10 for the operating system (effectively, the OS runs in hypervisor state) and MSR_{HV PR} = 11 for applications.

Reserved

Software must ensure that this bit contains 0; otherwise the results of executing all instructions are boundedly undefined.

Programming Note

This bit is initialized to 0 by hardware at system bringup. The handling of this bit by interrupts and by the rfid, hrfid, and rfscv instructions is such that, unless software deliberately sets the bit to 1, the bit will continue to contain 0.

Reserved

Transaction State (TS)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Non-transactional</td>
</tr>
<tr>
<td>01</td>
<td>Suspended</td>
</tr>
<tr>
<td>10</td>
<td>Transactional</td>
</tr>
<tr>
<td>11</td>
<td>Reserved</td>
</tr>
</tbody>
</table>
Changes to MSR\textsubscript{TS} that are caused by Transactional Memory instructions, and by invocation of the transaction’s failure handler, take effect immediately (even though these instructions and events are not context synchronizing).

31 \textbf{Transactional Memory Available (TM)}

0 The thread cannot execute any Transactional Memory instructions or access any Transactional Memory registers.
1 The thread can execute Transactional Memory instructions and access Transactional Memory registers unless the Transactional Memory facility has been made unavailable by some other register.

32:37 Reserved

38 \textbf{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.

39 Reserved

40 \textbf{VSX Available (VSX)}

0 The thread cannot execute any VSX instructions, including VSX loads, stores, and moves.
1 The thread can execute VSX instructions unless they have been made unavailable by some other register.

\textbf{Programming Note}

An application binary interface defined to support Vector-Scalar operations should also specify a requirement that MSR\textsubscript{FP} and MSR\textsubscript{VEC} be set to 1 whenever MSR\textsubscript{VSX} is set to 1.

41:47 Reserved

48 \textbf{External Interrupt Enable (EE)}

0 External, Decrementer, Performance Monitor, and Privileged Doorbell interrupts are disabled.
1 External, Decrementer, Performance Monitor, and Privileged Doorbell interrupts are enabled.

This bit also affects whether Hypervisor Decrementer, Hypervisor Maintenance, and Directed Hypervisor Doorbell interrupts are enabled; see Section 6.5.12 on page 1077, Section 6.5.19 on page 1086, and Section 6.5.20 on page 1086.

49 \textbf{Problem State (PR)}

0 The thread is in privileged state.
1 The thread is in problem state.

\textbf{Programming Note}

Any instruction that sets MSR\textsubscript{PR} to 1 also sets MSR\textsubscript{EE}, MSR\textsubscript{IR}, and MSR\textsubscript{DR} to 1.

50 \textbf{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.

\textbf{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 927.

\textbf{Programming Note}

The only instructions that can alter MSR\textsubscript{ME} are \texttt{rfid} and \texttt{hrfid}.

52 \textbf{Floating-Point Exception Mode 0 (FE0)}

See below.

53:54 \textbf{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 an \texttt{hrfid}, \texttt{rfid}, \texttt{rfscv}, or a Power-Saving Mode instruction, all of which are never traced. Successful completion means that the instruction caused no other interrupt and, if the processor is in the Transactional state, is not a disallowed instruction (e.g., \texttt{dcbf}) or an \texttt{mtspr} specifying an SPR that is not part of the checkpointed registers and is not the GSR (see Section 5.3.1 of Book II).
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 Instruction Relocate (IR)
0 Instruction address translation is disabled.
1 Instruction address translation is enabled.

Programming Note
See the Programming Note in the definition of MSRPR.

59 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.

Programming Note
See the Programming Note in the definition of MSRPR.

60 Reserved

61 Performance Monitor Mark (PMM)
This bit is used by software in conjunction with the Performance Monitor, as described in Chapter 9.

--- Programming Note ---
See the Programming Note in the definition of MSRPR.

62 Recoverable Interrupt (RI)
0 Interrupt is not recoverable.
1 Interrupt is recoverable.

Additional information about the use of this bit is given in Sections 6.4.3, “Interrupt Processing” on page 1059, 6.5.1, “System Reset Interrupt” on page 1065, and 6.5.2, “Machine Check Interrupt” on page 1067.

63 Little-Endian Mode (LE)
0 The thread is in Big-Endian mode.
1 The thread is in Little-Endian mode.

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>

3.2.2 State Transitions Associated with the Transactional Memory Facility

Updates to MSR_{TS} and MSR_{TM} caused by \texttt{rfebb, rfid, rfscv, hrfid,} or \texttt{mtmsrd} occur as described in Table 3. The value written, and whether or not the instruction causes an interrupt, are dependent on the current values of MSR_{TS} and MSR_{TM}, and the values being written to these fields. When the setting of MSR_{TS} causes an illegal state transition, a TM Bad Thing type Program interrupt is generated.

\begin{center}
\textbf{Programming Note}
\end{center}

The transition rules are the same for \texttt{mtmsrd} as for the \texttt{rfid}-type instructions because if a transition were illegal for \texttt{mtmsrd} but allowed for \texttt{rfid}, or vice versa, software could use the instruction for which the transition is allowed to achieve the effect of the other instruction.

Table 3 shows all the transaction state transitions that can be requested by \texttt{rfebb, rfid, rfscv, hrfid,} and \texttt{mtmsrd}. If PCR_{v2.06}=1 and the instruction requests a transition to problem state, transaction state transitions that the table shows as legal and as resulting in the thread being in Transactional or Suspended state instead cause a TM Bad Thing type Program interrupt; see Section 6.5.9. (The preceding sentence does not apply to \texttt{rfebb}, because \texttt{rfebb} cannot cause a change of privilege state, and cannot be executed in problem state when PCR_{v2.06}=1.) In the table, the contents of MSR_{TS} and MSR_{TM} are abbreviated in the form AB, where A represents MSR_{TS} (N, T or S) and B represents MSR_{TM} (0 or 1). “x” in the “B” position means that the entry covers both MSR_{TM} values, with the same value applying in all columns of a given row for a given instance of the transition. (E.g., the first row means that the transition from N0 to N0 is allowed and results in N0, and that the transition from N0 to N1 is allowed and results in N1.) “Input MSR_{TS} MSR_{TM}” in the second column refers to the MSR_{TS} and MSR_{TM} values supplied by CTR for \texttt{rfscv}, BESCR for \texttt{rfebb} (just the TS value), SRR1 for \texttt{rfid}, HSRR1 for \texttt{hrfid,} or register RS for \texttt{mtmsrd}.
### Table 3: Transaction state transitions that can be requested by `rfebb`, `rfid`, `rfscv`, `hrfid`, and `mtmsrd`.

<table>
<thead>
<tr>
<th>Current MSR$<em>{TS}$MSR$</em>{TM}$</th>
<th>Input MSR$<em>{TS}$MSR$</em>{TM}$</th>
<th>Resulting MSR$<em>{TS}$MSR$</em>{TM}$</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>N0</td>
<td>Nx</td>
<td>Nx</td>
<td>May occur in the context of a Transactional Memory type of Facility Unavailable interrupt handler, enabling/disabling transactions for user-level applications.</td>
</tr>
<tr>
<td>T0</td>
<td>N/A</td>
<td>N/A</td>
<td>Unreachable state</td>
</tr>
<tr>
<td>S0</td>
<td>N0$^2$</td>
<td>S0</td>
<td>Operating system code that is not TM aware may attempt to set TS and TM to zero, thinking they’re reserved bits. Change is suppressed.</td>
</tr>
<tr>
<td></td>
<td>T1</td>
<td>T1</td>
<td>May occur at an <code>rfid</code> returning to an application whose transaction was suspended on interrupt.</td>
</tr>
<tr>
<td></td>
<td>Sx</td>
<td>Sx</td>
<td>This case may occur for an <code>rfid</code> returning to an application whose suspended transaction was interrupted.</td>
</tr>
<tr>
<td></td>
<td>All others - Illegal$^1$</td>
<td>S0</td>
<td></td>
</tr>
<tr>
<td>N1</td>
<td>Nx</td>
<td>Nx</td>
<td>After a <code>treclaim</code>, the OS dispatches Nx program.</td>
</tr>
<tr>
<td></td>
<td>All others - Illegal$^1$</td>
<td>N0</td>
<td></td>
</tr>
<tr>
<td>T1</td>
<td>all</td>
<td>N1</td>
<td>Disallowed instructions in Transactional state</td>
</tr>
<tr>
<td>S1</td>
<td>T1</td>
<td>T1</td>
<td>May occur after <code>trechkpt</code> when returning to an application.</td>
</tr>
<tr>
<td></td>
<td>Sx</td>
<td>Sx</td>
<td></td>
</tr>
<tr>
<td></td>
<td>All others - Illegal$^1$</td>
<td>S0</td>
<td></td>
</tr>
</tbody>
</table>

**Notes:**

1. Generate TM Bad Thing type Program interrupt. “All others” includes all attempts to set MSR$_{TS}$ to 0b11 (reserved value).
2. Instruction completes, change to MSR$_{TM}$ suppressed, except when attempted by `rfebb`, in which case the result is a TM Bad Thing type Program interrupt.

---

1. Generate TM Bad Thing type Program interrupt. “All others” includes all attempts to set MSR$_{TS}$ to 0b11 (reserved value).
2. Instruction completes, change to MSR$_{TM}$ suppressed, except when attempted by `rfebb`, in which case the result is a TM Bad Thing type Program interrupt.
Programming Note

For rfscv, [hjrfd], and mtmsrd, the attempted transition from S0 to N0 is suppressed in order that interrupt handlers that are "unaware" of transactional memory, and load an MSR value that has not been updated to take account of transactional memory, will continue to work correctly. (If the interrupt occurs when a transaction is running or suspended, the interrupt will set MSR\textsubscript{TS||TM} to S0. If the interrupt handler attempts to load an MSR value that has not been updated to take account of transactional memory, that MSR value will have TS || TM = N0. It is desirable that the interrupt handler remain in state S0, so that it can return normally to the interrupted transaction.)

The problem solved by suppressing this transition does not apply to rfebb, so for rfebb an attempt to transition from S0 to N0 is not suppressed, and instead causes a TM Bad Thing type Program interrupt.
3.2.3 Processor Stop Status and Control Register (PSSCR)

The layout of the PSSCR is shown below.

![Figure 6. Processor stop Status and Control Register](image)

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**

Before dispatching an OS, the hypervisor may initialize this field to 1 in order to prevent the OS from reading the Power-Saving Level Status (PLS) field. This may be necessary in secure environments since an OS may be capable of detecting the presence of another OS on the same processor by observing the state of the PLS field after exiting power-saving mode.

42 **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 6.5.26.

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.
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  Provided LPCR_{PECE} is not lost, hardware will exit power-saving mode only 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.

Programming Note

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 3.3.2.

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.

48:53  Reserved

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.

56:59  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 6.5.26 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 during power-saving mode beyond the value indicated in the RL field.

Requested Level (RL)

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 6.5.26.

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 6.5.26 for details.
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 `PSSCRRL > PSSCRPSLL` 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 `PSSCRMTL` to a value that causes state loss, or setting both `PSSCRRL` and `PSSCRMTL` 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.
3.3 Branch Facility Instructions

3.3.1 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>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td>16</td>
<td>15</td>
<td>14</td>
<td>13</td>
<td>12</td>
<td>11</td>
<td>10</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 6.5, "Interrupt Definitions" on page 1063. The setting of the MSR is affected by the contents of the LEV field. LEV values greater than 1 are reserved. Bits 0:5 of the LEV field (instruction bits 20:25) 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. This is the only way that executing an instruction can cause hypervisor state to be entered.

Because this instruction is not privileged, it is possible for application software to invoke the hypervisor. However, such invocation 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

17  //  11  //  LEV  //  0  //  1

LR ← CIA + 4
CTR33:36 42:47 ← undefined
CTR0:32 37:41 48:63 ← MSR0:32 37:41 48:63
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 6.5.

The interrupt causes the next instruction to be fetched as specified in LPCRAll (see to 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**

```
rfserv

0  19  //  11  //  16  //  21  //  82  //  31

if (MSR29:31 ≠ 0b010 | CTR29:31 ≠ 0b000) then
    MSR29:31 ← CTR29:31
    MSR48 ← CTR48  |  CTR49
    MSR58 ← CTR58  |  CTR49
    MSR59 ← CTR59  |  CTR49
    NIA ← LR0:61 || 0b00
```

If bits 29 through 31 of the MSR are not equal to 0b010 or bits 29 through 31 of the Count Register are not equal to 0b000, then the value of bits 29 through 31 of the Count Register is placed into bits 29 through 31 of the MSR. The result of ORing bits 48 and 49 of the Count Register is placed into MSR48. The result of ORing bits 58 and 49 of the Count Register is placed into MSR58. The result of ORing bits 59 and 49 of the Count Register is placed into MSR59. Bits 0:2, 4:28, 32, 37:41, 49:50, 52:57, and 60:63 of the Count Register are placed into the corresponding bits of the MSR.

If the instruction attempts to cause an illegal transaction state transition or, when TM is made unavailable in problem state by the PCR, attempts to cause a transition to problem state and also a transaction state transition that Table 3 on page 947 shows as legal and as resulting in the thread being in Transactional or Suspended state, a TM Bad Thing type Program interrupt is generated (unless a higher-priority exception is pending). If this interrupt is generated, the value placed into SRR0 by the interrupt processing mechanism (see Section 6.4.3) is the address of the *rfscv* instruction. Otherwise, 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 LR0:61 || 0b00 (when SF=1 in the new MSR value) or 320 || LR32:61 || 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 or HSRR0 by the interrupt processing mechanism (see Section 6.4.3) 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 MSRER, MSHR, and MSRD to 1.
**Return From Interrupt Doubleword**

**XL-form**

<table>
<thead>
<tr>
<th></th>
<th>19</th>
<th>16</th>
<th>13</th>
<th>10</th>
<th>8</th>
<th>6</th>
<th>3</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>rfid</td>
<td>///</td>
<td>///</td>
<td>///</td>
<td>18</td>
<td>/</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

If MSR$_3$=1 then bits 3 and 51 of SRR1 are placed into the corresponding bits of the MSR. If bits 29 through 31 of the MSR are not equal to 0b010 or bits 29 through 31 of SRR1 are not equal to 0b000, then the value of bits 29 through 31 of SRR1 is placed into bits 29 through 31 of the MSR. The result of ORing bits 48 and 49 of SRR1 is placed into MSR$_{48}$. The result of ORing bits 58 and 49 of SRR1 is placed into MSR$_{58}$. The result of ORing bits 59 and 49 of SRR1 is placed into MSR$_{59}$. Bits 0:2, 4:28, 32, 37:41, 49:50, 52:57, and 60:63 of SRR1 are placed into the corresponding bits of the MSR.

If the instruction attempts to cause an illegal transaction state transition or, when TM is made unavailable in problem state by the PCR, attempts to cause a transition to problem state and also a transaction state transition that Table 3 on page 947 shows as legal and as resulting in the thread being in Transactional or Suspended state, a TM Bad Thing type Program interrupt is generated (unless a higher-priority exception is pending). If this interrupt is generated, the value placed into SRR0 by the interrupt processing mechanism (see Section 6.4.3) is the address of the `rfid` instruction. Otherwise, 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$_{32:61}$ || 0b00 (when SF=1 in the new MSR value) or 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 or HSRR0 by the interrupt processing mechanism (see Section 6.4.3) 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.
Hypervisor Return From Interrupt Doubleword  XL-form

hrfid

```
  0  6  11  16  21  274  31
```

If bits 29 through 31 of the MSR are not equal to 0b010 or bits 29 through 31 of HSRR1 are not equal to 0b000, then the value of bits 29 through 31 of HSRR1 is placed into bits 29 through 31 of the MSR. The result of ORing bits 48 and 49 of HSRR1 is placed into MSR48. The result of ORing bits 58 and 49 of HSRR1 is placed into MSR58. The result of ORing bits 59 and 49 of HSRR1 is placed into MSR59. Bits 0:28, 32, 37:41, 49:57, and 60:63 of HSRR1 are placed into the corresponding bits of the MSR.

If the instruction attempts to cause an illegal transaction state transition or, when TM is made unavailable in problem state by the PCR, attempts to cause a transition to problem state and also a transaction state transition that Table 3 on page 947 shows as legal and as resulting in the thread being in Transactional or Suspended state, a TM Bad Thing type Program interrupt is generated (unless a higher-priority exception is pending). If this interrupt is generated, the value placed into SRR0 by the interrupt processing mechanism (see Section 6.4.3) is the address of the hrfid instruction. Otherwise, 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 HSRR028:31 || 0b00 (when SF=1 in the new MSR value) or 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 or HSRR0 by the interrupt processing mechanism (see Section 6.4.3) 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 MSR_{EE}, MSR_{IR}, and MSR_{DR} to 1.
3.3.2 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 $\text{PSSCR}_{\text{ESL}} = 0$ 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 and the HRMOR 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.

The thread can be put in power-saving mode by executing the `stop` instruction. As specified below, this instruction stops execution immediately after the stop instruction is executed, and the thread is put into power-saving mode. The power-saving level that is entered depends on the contents of the PSSCR (see Section 3.2.3).
### 3.3.2.1 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 3.2.3.)

#### 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 3.2.3). 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 3.3.2. (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 PSSCRRML must be set to values that do not allow state loss. (See the definition of the EC bit description in Section 3.3.2.) This guarantees that the state of MSREE is not lost.

If stop was executed when PSSCREC=0 and MSREE=0 (in order to avoid the hang condition described in the above 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.

The thread remains in power-saving mode until either a System Reset exception or certain other events occur. The events that may cause exit from power-saving mode are specified by PSSCREC and LPCRPECE. If the event that causes the exit is a System Reset, Machine Check, or Hypervisor Maintenance exception, resource state that would be lost if the exception occurred when the thread was not in power-saving mode may be lost.

An attempt to execute this instruction in Suspended state will result in a TM Bad Thing type Program interrupt.

This instruction is privileged and context synchronizing.

**Special Registers Altered:**

None

### 3.3.2.2 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 3.2.3 for additional information.

If the *stop* instruction is to be executed by the hypervisor when PSSCREC=1, the LPCRPECE must be set to the desired value (see Section 2.2). 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 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 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 MSREE=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 After the thread has entered power-saving mode with PSSCREC=1, only the System Reset or Machine Check exceptions and the exceptions enabled in LPCRPECE 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 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.
3.4 Event-Based Branch Facility and Instruction

The Event-Based Branch facility is described in Chapter 7 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 BESCGRGE=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 BESCPRPME must be set to 1 to enable Performance Monitor event-based exceptions.
- BESCREE must be set to 1 to enable External event-based exceptions.

If an event-based exception exists (as indicated by BESCPRPEMO=1 or BESCREEEO=1) when MSRPR=0, the corresponding event-based branch will occur when MSRPR=1, FSCR EBB=1, HFSCR EBB=1, and BESCGRGE=1.

Software EBB handlers should ensure that previous exceptions have been cleared (by setting BESCPRPEMO and/or BESCREEEO to 0) before re-enabling event-based branches (by setting BESCGRGE to 1 or executing rfebb 1) in order to prevent earlier exceptions from causing additional EBBs.

If the rfebb instruction attempts to cause an illegal transaction state transition (see Section 3.2.2), a TM Bad Thing type Program interrupt is generated (unless a higher-priority exception is pending). If this interrupt is generated, the value placed into SRR0 by the interrupt processing mechanism is the address of the rfebb instruction.
Chapter 4. Fixed-Point Facility

4.1 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.

4.2 Special Purpose Registers

Special Purpose Registers (SPRs) are read and written using the `mfspr` (page 975) and `mtspr` (page 974) instructions. Most SPRs are defined in other chapters of this book; see the index to locate those definitions.

4.3 Fixed-Point Facility Registers

4.3.1 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 `mfspr` instruction. Read access to the PVR is privileged; write access is not provided.

**Figure 7. Processor Version Register**

The PVR distinguishes between implementations that differ in attributes that may affect software. It contains two fields.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>32:48</td>
<td>Version</td>
</tr>
<tr>
<td>63</td>
<td>Revision</td>
</tr>
</tbody>
</table>

**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.

4.3.2 Chip Information Register

The Chip Information Register (CIR) is a 32-bit read-only register that contains a value identifying the manufacturer and other characteristics of the chip on which the processor is implemented. The contents of the CIR can be copied to a GPR by the `mfspr` instruction. Read access to the CIR is privileged; write access is not provided.

**Figure 8. Chip Information Register**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>32:36</td>
<td>Manufacturer ID (ID)</td>
</tr>
<tr>
<td>63</td>
<td>Implementation-dependent.</td>
</tr>
</tbody>
</table>

**Bit  Description**

| 32:35 | Manufacturer ID (ID) A four-bit field that identifies the manufacturer of the chip. |
| 36:63 | Implementation-dependent. |

4.3.3 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 `mfspr` instruction. Read access to the PIR is privileged; write access is not provided.
4.3.4 Process Identification Register

The layout of the Process Identification Register (PIDR) is shown in Figure 10 below.

![Figure 10. Process Identification Register](image)

<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>32:43</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 MSR_{PR}=0.

4.3.5 Thread ID Register

The Thread ID Register (TIDR) is a 64-bit register that holds an identifier for the thread that is unique among threads with the same Process ID that are using accelerators. The layout of the Thread Identification Register (TIDR) is shown in Figure 11 below.

![Figure 11. Thread Identification Register](image)

<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:63</td>
<td>TID</td>
<td>Thread Identifier</td>
</tr>
</tbody>
</table>

An implementation may opt to implement only the least-significant \( n \) bits of the Thread ID Register, where \( 0 \leq n \leq 64 \). The most-significant \( 64-n \) bits of the Thread ID Register are treated as reserved.

Access to the TIDR is privileged.

**Programming Note**

The TIDR is used by platform hardware to deliver a notification signal that will complete wait on the appropriate thread. This “platform notify” signal commonly reports the completion of processing by an accelerator. See Section 4.6.4, “Wait Instruction”, in Book II for additional details. See platform documentation for possible synchronization requirements for changing the TID.

4.3.6 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>
<tr>
<td>56:62</td>
<td>Reserved</td>
</tr>
<tr>
<td>63</td>
<td>RUN</td>
</tr>
</tbody>
</table>

**Problem State Access**

Privileged Non-hypervisor State Access

Bits 0:7 of this field are read-only bits that indicate the state of CTRL_{RUN} for threads with privileged thread numbers 0 through 7, respectively; bits corresponding to privileged thread numbers higher than the maximum privileged thread number supported are set to 0s.

Hypervisor State Access

Bits 0:7 of this field are read-only bits that indicate the state of CTRL_{RUN} for threads with hypervisor thread numbers 0 through 7, respectively; bits corresponding to hypervisor thread numbers higher than the maximum hypervisor thread number supported are set to 0s.

56:62 | Reserved |
63 | RUN |

This bit controls an external I/O pin. This signal may be used for the following:
driving the RUN Light on a system operator panel
Direct External exception routing
Performance Monitor Counter incrementing (see Chapter 9)

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.

### 4.3.7 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 4.3.8) 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>
<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>
<tr>
<td>110</td>
<td>high</td>
</tr>
<tr>
<td>111</td>
<td>very high</td>
</tr>
</tbody>
</table>

#### 4.3.8 Problem State Priority Boost Register

The Problem State Priority Boost (PSPB) register is a 32-bit register that controls whether problem state programs have access to program priority medium high. (See Section 3.1 of Book II.)

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 (MSRPR=1).

#### 4.3.9 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 PPRPRI.

<table>
<thead>
<tr>
<th>Bits</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:1</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

Each RPn field is defined as follows.

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2:7</td>
<td>Relative priority of priority level n: Specifies the relative priority that corresponds to PPRPRI=n, where a value of 0 indicates the lowest relative priority and a value of 0b111111 indicates the highest relative priority.</td>
</tr>
</tbody>
</table>

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.

---

**Programming 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.
4.3.10 Software-use SPRs

Software-use SPRs are 64-bit registers provided for use by software.

<table>
<thead>
<tr>
<th>SPRG0</th>
<th>SPRG1</th>
<th>SPRG2</th>
<th>SPRG3</th>
</tr>
</thead>
<tbody>
<tr>
<td></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 \textit{mfspr} instruction.

<table>
<thead>
<tr>
<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>
</tr>
</tbody>
</table>

Programming Note

Neither the contents of the SPRGs, nor accessing them using \textit{mfspr} or \textit{mfspr}, has a side effect on the operation of the thread. One or more of the registers is likely to be needed by non-hypervisor interrupt handler programs (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>HSPRG0</th>
<th>HSPRG1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>63</td>
</tr>
</tbody>
</table>

Figure 16. SPRs for use by hypervisor programs

Programming Note

Neither the contents of the HSPRGs, nor accessing them using \textit{mfspr} or \textit{mfspr}, has a side effect on the operation of the thread. One or more of the registers is likely to be needed by hypervisor interrupt handler programs (e.g., as scratch registers and/or pointers to per thread save areas).
4.4 Fixed-Point Facility Instructions

4.4.1 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 Died=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

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

<table>
<thead>
<tr>
<th>31</th>
<th>21</th>
<th>16</th>
<th>11</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>853</td>
</tr>
<tr>
<td></td>
<td>31</td>
<td></td>
<td></td>
<td>/</td>
</tr>
</tbody>
</table>

if RA = 0 then b ← 0
else b ← (RA)
EA ← b + (RB)
RT ← 560 || MEM(EA, 1)

Let the effective address (EA) be the sum (RA|0)+ (RB). The byte in storage addressed by EA is loaded into RT56:63. RT0:55 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

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

<table>
<thead>
<tr>
<th>31</th>
<th>21</th>
<th>16</th>
<th>11</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>821</td>
</tr>
<tr>
<td></td>
<td>31</td>
<td></td>
<td></td>
<td>/</td>
</tr>
</tbody>
</table>

if RA = 0 then b ← 0
else b ← (RA)
EA ← b + (RB)
RT ← 480 || 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.

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

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

<table>
<thead>
<tr>
<th>31</th>
<th>21</th>
<th>16</th>
<th>11</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>789</td>
</tr>
<tr>
<td></td>
<td>31</td>
<td></td>
<td></td>
<td>/</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.

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

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

<table>
<thead>
<tr>
<th>31</th>
<th>21</th>
<th>16</th>
<th>11</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>885</td>
</tr>
<tr>
<td></td>
<td>31</td>
<td></td>
<td></td>
<td>/</td>
</tr>
</tbody>
</table>

if RA = 0 then b ← 0
else b ← (RA)
EA ← b + (RB)
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.

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**

stbcix    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 ← 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.

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**

sthcix    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 ← 0
else b ← (RA)
EA ← b + (RB)
MEM(EA, 2) ← (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**

stwcix    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 ← 0
else b ← (RA)
EA ← b + (RB)
MEM(EA, 4) ← (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**

stdcix    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 ← 0
else b ← (RA)
EA ← b + (RB)
MEM(EA, 8) ← (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
4.4.2 OR Instruction

*or Rx,Rx,Rx* can be used to set $PPR_{PRI}$ (see Section 4.3.7) as shown in Figure 17. For all priorities except medium high, $PPR_{PRI}$ 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, $PPR_{PRI}$ 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>$PPR_{PRI}$</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/yes$^1$</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>

$^1$This value is privileged unless the Problem State Priority Boost register allows the priority value 0b101 (See Section 4.3.8.)

Figure 17. Priority levels for *or Rx,Rx,Rx*
### 4.4.3 Transactional Memory Instructions

Privileged software that makes the Transactional Memory Facility available to applications takes on the responsibility of managing the facility's resources and the application's transaction state during interrupt handling, service calls, task switches, and its own use of TM. In addition to the existing instructions like `rfid` and problem state TM instructions that play a role in this management, `treclaim` and `trechkpt` may be used, as described below. See Section 3.2.2 for additional information about managing the TM facility and associated state transitions.

**Transaction Reclaim X-form**

```
Treclaim. RA

| 31 | 6 | 11 | 16 | 21 | 942 | 1 |
```

| CR0 ← 0 || MSRtsg || 0 |
| if MSRtsg = 0b10 | MSRtsg = 0b01 then |
| #Transactional or Suspended |
| if RA = 0 then cause ← 0x00000001 |
| else cause ← GPR(RA)56:63 || 0x00000001 |
| if TEXASRtsg = 0 then |
| Discard transactional footprint |
| TMRecordFailure(cause) |
| checkpointed registers ← |
| contents from checkpoint area |
| Discard all resources related to current trans-
action |
| MSRtsg ← 0b00 #Non-transactional |

The `treclaim` instruction frees the transactional facility for use by a new transaction. It sets condition register field 0 to 0 || MSRtsg || 0. If the transactional facility is in the Transactional state or Suspended state, failure recording is performed as defined in Section 5.3.2 of Book II. If RA is 0, the failure cause is set to 0x00000001, otherwise it is set to GPR(RA)56:63 || 0x00000001. The checkpointed registers are restored from the checkpoint area, and all resources related to the current transaction are discarded, including the transactional footprint (if it wasn’t already discarded for a pending failure).

The transaction state is set to Non-transactional.

If an attempt is made to execute `treclaim` in Non-transactional state, a TM Bad Thing type Program interrupt will be generated.

This instruction is privileged.

**Special Registers Altered:**

CR0TEXASR TFIAR TS

---

**Programming Note**

The `treclaim` instruction can be used by an interrupt handler to deallocate the current thread’s transactional resources in preparation for subsequent use of the facility by a new transaction. (An abort is not appropriate for this use, because (a) the interrupt handler is in Suspended state and an abort in Suspended state leaves the thread in Suspended state, and (b) an abort in Suspended state does not restore the checkpointed registers.) After `treclaim` is executed, the interrupt handler should save the contents of the checkpointed registers to storage. When the interrupted program is next dispatched it should be resumed by first restoring the contents of the checkpointed registers from storage and then using `trechkpt` to copy the contents of the checkpointed registers to the checkpoint area. (This saving and restoring of the checkpointed register state is in addition to the normal saving and restoring of the entire current register state.) The result of this use of `treclaim` and `trechkpt` is to restore the pre-transactional register values into the checkpoint area and to cause the thread to transition from Suspended state to Non-transactional state and back again. Failure handling for the program will occur when the program next attempts to execute an instruction in the Transactional state, which will cause the failure handler to be invoked because TDOOMED will be 1. (This will be immediate if the program was in the Transactional state when the interrupt occurred, or will be after `tresume` is executed if the program was in the Suspended state when the interrupt occurred.)
Transaction Recheckpoint X-form

trechkpt.

<table>
<thead>
<tr>
<th>0</th>
<th>31</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>1006</th>
<th>1</th>
</tr>
</thead>
</table>

CR0 ← 0 || MSR_T || 0
MSR_T ← 0b01
TDOOMED ← 1
checkpoint area ← {checkpointed registers}

The `trechkpt.` instruction copies the contents of the checkpointed registers to the checkpoint area. It sets condition register field 0 to 0 || MSR_T || 0. The current values of the checkpointed registers are loaded into the checkpoint area. TDOOMED is set to 0b1.

The transaction state is set to Suspended.

If an attempt is made to execute this instruction in Transactional or Suspended state or when TEXAS-RFS=0, a TM Bad Thing type Program interrupt will be generated.

This instruction is privileged.

Special Registers Altered:

CR0 TS

4.4.4 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 18 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 158, identified in Figure 18 as GSR, is a special SPR in that it retains no state and exists only to identify a performance optimization opportunity. `mtspr` specifying the GSR (Group Start Register) is used to identify the start of a sequence of `mtspr` instructions that may be optimized to have their SPR changes synchronized once as a group, rather than independently. The sequence is ended by any instruction other than a `mtspr` and also by an implicit redirection of instruction fetching, including those caused by interrupts and transaction failure. This function may be useful when restoring a number of SPRs. If any of the `mtspr` instructions in the sequence requires explicit context synchronization, a context synchronizing instruction must follow the sequence. See Chapter 11 of Book III for more details. Write access to the GSR is privileged; read access is not provided.

SPR numbers that are not shown in Figure 18 and are in the ranges shown below are reserved for implementation-specific uses.

848 - 863
880 - 895
976 - 991
1008 - 1023

Implementation-specific registers must be privileged. SPR numbers for implementation-specific SPRs should be registered in advance with the Power ISA architects.
### Figure 18. SPR encodings (Sheet 1 of 3)

<table>
<thead>
<tr>
<th>decimal</th>
<th>SPR&lt;sup&gt;†&lt;/sup&gt;</th>
<th>Register Name</th>
<th>Privileged</th>
<th>Length (bits)</th>
<th>Extended Mnemonics&lt;sup&gt;‡&lt;/sup&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>spr&lt;sub&gt;5:0&lt;/sub&gt;, spr&lt;sub&gt;0:4&lt;/sub&gt;</td>
<td></td>
<td>mtspr</td>
<td>mfspr</td>
<td>mtspr</td>
</tr>
<tr>
<td>1</td>
<td>00000 00001</td>
<td>XER</td>
<td>no</td>
<td>no</td>
<td>64</td>
</tr>
<tr>
<td>3</td>
<td>00000 00011</td>
<td>DSCR</td>
<td>no</td>
<td>no</td>
<td>64</td>
</tr>
<tr>
<td>8</td>
<td>00000 01000</td>
<td>LR</td>
<td>no</td>
<td>no</td>
<td>64</td>
</tr>
<tr>
<td>9</td>
<td>00000 01001</td>
<td>CTR</td>
<td>no</td>
<td>no</td>
<td>64</td>
</tr>
<tr>
<td>13</td>
<td>00000 01101</td>
<td>AMR</td>
<td>no&lt;sup&gt;4&lt;/sup&gt;</td>
<td>no</td>
<td>64</td>
</tr>
<tr>
<td>17</td>
<td>00000 10001</td>
<td>DSCR</td>
<td>yes</td>
<td>yes</td>
<td>32</td>
</tr>
<tr>
<td>18</td>
<td>00000 10010</td>
<td>DSISR</td>
<td>yes</td>
<td>yes</td>
<td>32</td>
</tr>
<tr>
<td>19</td>
<td>00000 10011</td>
<td>DAR</td>
<td>yes</td>
<td>yes</td>
<td>64</td>
</tr>
<tr>
<td>22</td>
<td>00000 10110</td>
<td>DEC</td>
<td>yes</td>
<td>yes</td>
<td>64</td>
</tr>
<tr>
<td>26</td>
<td>00000 11010</td>
<td>SSR0</td>
<td>yes</td>
<td>yes</td>
<td>64</td>
</tr>
<tr>
<td>27</td>
<td>00000 11101</td>
<td>SSR1</td>
<td>yes</td>
<td>yes</td>
<td>64</td>
</tr>
<tr>
<td>48</td>
<td>00001 10000</td>
<td>PISR</td>
<td>yes&lt;sup&gt;13&lt;/sup&gt;</td>
<td>yes</td>
<td>32</td>
</tr>
<tr>
<td>61</td>
<td>00011 11101</td>
<td>IAMR</td>
<td>yes</td>
<td>yes</td>
<td>64</td>
</tr>
<tr>
<td>128</td>
<td>00100 00000</td>
<td>TPHAR</td>
<td>no</td>
<td>no</td>
<td>64</td>
</tr>
<tr>
<td>129</td>
<td>00100 00001</td>
<td>TFIAR</td>
<td>no</td>
<td>no</td>
<td>64</td>
</tr>
<tr>
<td>130</td>
<td>00100 00010</td>
<td>TEXASR</td>
<td>no</td>
<td>no</td>
<td>64</td>
</tr>
<tr>
<td>131</td>
<td>00100 00011</td>
<td>TEXASRU</td>
<td>no</td>
<td>no</td>
<td>32</td>
</tr>
<tr>
<td>136</td>
<td>00100 01000</td>
<td>CTRL</td>
<td>-</td>
<td>no</td>
<td>32</td>
</tr>
<tr>
<td>144</td>
<td>00100 10000</td>
<td>TIDR</td>
<td>yes</td>
<td>yes</td>
<td>64</td>
</tr>
<tr>
<td>152</td>
<td>00100 11000</td>
<td>CTRL</td>
<td>yes</td>
<td>yes</td>
<td>32</td>
</tr>
<tr>
<td>153</td>
<td>00100 11001</td>
<td>FSSCR</td>
<td>yes</td>
<td>yes</td>
<td>64</td>
</tr>
<tr>
<td>157</td>
<td>00100 11101</td>
<td>UAMOR</td>
<td>yes&lt;sup&gt;5&lt;/sup&gt;</td>
<td>yes</td>
<td>64</td>
</tr>
<tr>
<td>158</td>
<td>00100 11110</td>
<td>GSR</td>
<td>yes</td>
<td>-</td>
<td>9</td>
</tr>
<tr>
<td>159</td>
<td>00100 11111</td>
<td>PSPB</td>
<td>yes</td>
<td>yes</td>
<td>32</td>
</tr>
<tr>
<td>176</td>
<td>00101 10000</td>
<td>DPDES</td>
<td>hypv&lt;sup&gt;2&lt;/sup&gt;</td>
<td>yes</td>
<td>64</td>
</tr>
<tr>
<td>180</td>
<td>00101 10100</td>
<td>DAWR0</td>
<td>hypv&lt;sup&gt;2&lt;/sup&gt;</td>
<td>hypv&lt;sup&gt;2&lt;/sup&gt;</td>
<td>64</td>
</tr>
<tr>
<td>186</td>
<td>00101 11010</td>
<td>RPR</td>
<td>hypv&lt;sup&gt;2&lt;/sup&gt;</td>
<td>hypv&lt;sup&gt;2&lt;/sup&gt;</td>
<td>64</td>
</tr>
<tr>
<td>187</td>
<td>00101 11011</td>
<td>CIABR</td>
<td>hypv&lt;sup&gt;2&lt;/sup&gt;</td>
<td>hypv&lt;sup&gt;2&lt;/sup&gt;</td>
<td>64</td>
</tr>
<tr>
<td>188</td>
<td>00101 11100</td>
<td>DAWRX0</td>
<td>hypv&lt;sup&gt;2&lt;/sup&gt;</td>
<td>hypv&lt;sup&gt;2&lt;/sup&gt;</td>
<td>32</td>
</tr>
<tr>
<td>190</td>
<td>00101 11110</td>
<td>HFSCR</td>
<td>hypv&lt;sup&gt;2&lt;/sup&gt;</td>
<td>hypv&lt;sup&gt;2&lt;/sup&gt;</td>
<td>64</td>
</tr>
<tr>
<td>256</td>
<td>00100 00000</td>
<td>VRSAVE</td>
<td>no</td>
<td>no</td>
<td>32</td>
</tr>
<tr>
<td>259</td>
<td>00100 00011</td>
<td>SPRG3</td>
<td>-</td>
<td>no</td>
<td>64</td>
</tr>
<tr>
<td>268</td>
<td>00100 01100</td>
<td>TB</td>
<td>-</td>
<td>no</td>
<td>64</td>
</tr>
<tr>
<td>269</td>
<td>00100 01101</td>
<td>TBU</td>
<td>-</td>
<td>no</td>
<td>32</td>
</tr>
<tr>
<td>272-275</td>
<td>00100 100xx</td>
<td>SPRG&lt;sub&gt;n&lt;/sub&gt;</td>
<td>yes</td>
<td>yes</td>
<td>64</td>
</tr>
<tr>
<td>283</td>
<td>00100 11101</td>
<td>CIR</td>
<td>-</td>
<td>yes</td>
<td>32</td>
</tr>
<tr>
<td>284</td>
<td>00100 11110</td>
<td>TBL</td>
<td>hypv&lt;sup&gt;2&lt;/sup&gt;</td>
<td>-</td>
<td>32</td>
</tr>
<tr>
<td>285</td>
<td>00100 11111</td>
<td>TBU</td>
<td>hypv&lt;sup&gt;2&lt;/sup&gt;</td>
<td>-</td>
<td>32</td>
</tr>
<tr>
<td>286</td>
<td>00100 11110</td>
<td>TBU40</td>
<td>hypv&lt;sup&gt;2&lt;/sup&gt;</td>
<td>-</td>
<td>64</td>
</tr>
<tr>
<td>287</td>
<td>00100 11111</td>
<td>PVR</td>
<td>-</td>
<td>yes</td>
<td>32</td>
</tr>
<tr>
<td>304</td>
<td>00101 10000</td>
<td>HSPPRGR0</td>
<td>hypv&lt;sup&gt;2&lt;/sup&gt;</td>
<td>hypv&lt;sup&gt;2&lt;/sup&gt;</td>
<td>64</td>
</tr>
<tr>
<td>305</td>
<td>00101 10001</td>
<td>HSPPRGR1</td>
<td>hypv&lt;sup&gt;2&lt;/sup&gt;</td>
<td>hypv&lt;sup&gt;2&lt;/sup&gt;</td>
<td>64</td>
</tr>
<tr>
<td>306</td>
<td>00101 10010</td>
<td>HDSISR</td>
<td>hypv&lt;sup&gt;2&lt;/sup&gt;</td>
<td>hypv&lt;sup&gt;2&lt;/sup&gt;</td>
<td>32</td>
</tr>
<tr>
<td>307</td>
<td>00101 10011</td>
<td>HDRAR</td>
<td>hypv&lt;sup&gt;2&lt;/sup&gt;</td>
<td>hypv&lt;sup&gt;2&lt;/sup&gt;</td>
<td>64</td>
</tr>
<tr>
<td>308</td>
<td>00101 10100</td>
<td>SPURR</td>
<td>hypv&lt;sup&gt;2&lt;/sup&gt;</td>
<td>yes</td>
<td>64</td>
</tr>
<tr>
<td>309</td>
<td>00101 10101</td>
<td>PURR</td>
<td>hypv&lt;sup&gt;2&lt;/sup&gt;</td>
<td>yes</td>
<td>64</td>
</tr>
<tr>
<td>310</td>
<td>00101 10110</td>
<td>HDEC</td>
<td>hypv&lt;sup&gt;2&lt;/sup&gt;</td>
<td>hypv&lt;sup&gt;2&lt;/sup&gt;</td>
<td>64</td>
</tr>
<tr>
<td>313</td>
<td>00101 11001</td>
<td>HRMOR</td>
<td>hypv&lt;sup&gt;2&lt;/sup&gt;</td>
<td>hypv&lt;sup&gt;2&lt;/sup&gt;</td>
<td>64</td>
</tr>
<tr>
<td>314</td>
<td>00101 11010</td>
<td>HSRR0</td>
<td>hypv&lt;sup&gt;2&lt;/sup&gt;</td>
<td>hypv&lt;sup&gt;2&lt;/sup&gt;</td>
<td>64</td>
</tr>
<tr>
<td>315</td>
<td>00101 11011</td>
<td>HSRR1</td>
<td>hypv&lt;sup&gt;2&lt;/sup&gt;</td>
<td>hypv&lt;sup&gt;2&lt;/sup&gt;</td>
<td>64</td>
</tr>
</tbody>
</table>
### Figure 18. SPR encodings (Sheet 2 of 3)

<table>
<thead>
<tr>
<th>decimal</th>
<th><strong>SPR</strong></th>
<th>Privileged</th>
<th>Length</th>
<th>Extended Mnemonics*</th>
</tr>
</thead>
<tbody>
<tr>
<td>318</td>
<td>sp5:9:4</td>
<td>LPCR</td>
<td>hypv2</td>
<td>64</td>
</tr>
<tr>
<td>319</td>
<td>sp5:9:4</td>
<td>LPIDR</td>
<td>hypv2</td>
<td>64</td>
</tr>
<tr>
<td>336</td>
<td>sp5:9:4</td>
<td>HMER</td>
<td>hypv2</td>
<td>64</td>
</tr>
<tr>
<td>337</td>
<td>sp5:9:4</td>
<td>HMEER</td>
<td>hypv2</td>
<td>64</td>
</tr>
<tr>
<td>338</td>
<td>sp5:9:4</td>
<td>PCR</td>
<td>hypv2</td>
<td>64</td>
</tr>
<tr>
<td>339</td>
<td>sp5:9:4</td>
<td>HEIR</td>
<td>hypv2</td>
<td>64</td>
</tr>
<tr>
<td>349</td>
<td>sp5:9:4</td>
<td>AMOR</td>
<td>hypv2</td>
<td>64</td>
</tr>
<tr>
<td>446</td>
<td>sp5:9:4</td>
<td>TIR</td>
<td>yes</td>
<td>64</td>
</tr>
<tr>
<td>464</td>
<td>sp5:9:4</td>
<td>PTCR</td>
<td>hypv2</td>
<td>64</td>
</tr>
<tr>
<td>68</td>
<td>sp5:9:4</td>
<td>SIER</td>
<td>-</td>
<td>64</td>
</tr>
<tr>
<td>70</td>
<td>sp5:9:4</td>
<td>MMCRA</td>
<td>no6</td>
<td>64</td>
</tr>
<tr>
<td>71</td>
<td>sp5:9:4</td>
<td>PMC1</td>
<td>no6</td>
<td>32</td>
</tr>
<tr>
<td>72</td>
<td>sp5:9:4</td>
<td>PMC2</td>
<td>no6</td>
<td>32</td>
</tr>
<tr>
<td>73</td>
<td>sp5:9:4</td>
<td>PMC3</td>
<td>no6</td>
<td>32</td>
</tr>
<tr>
<td>74</td>
<td>sp5:9:4</td>
<td>PMC4</td>
<td>no6</td>
<td>32</td>
</tr>
<tr>
<td>75</td>
<td>sp5:9:4</td>
<td>PMC5</td>
<td>no6</td>
<td>32</td>
</tr>
<tr>
<td>76</td>
<td>sp5:9:4</td>
<td>PMC6</td>
<td>no6</td>
<td>32</td>
</tr>
<tr>
<td>77</td>
<td>sp5:9:4</td>
<td>MMCR0</td>
<td>no6</td>
<td>64</td>
</tr>
<tr>
<td>78</td>
<td>sp5:9:4</td>
<td>SIAR</td>
<td>-</td>
<td>64</td>
</tr>
<tr>
<td>79</td>
<td>sp5:9:4</td>
<td>SDAI</td>
<td>-</td>
<td>64</td>
</tr>
<tr>
<td>80</td>
<td>sp5:9:4</td>
<td>MMCR1</td>
<td>-</td>
<td>64</td>
</tr>
<tr>
<td>84</td>
<td>sp5:9:4</td>
<td>SIER</td>
<td>yes</td>
<td>64</td>
</tr>
<tr>
<td>85</td>
<td>sp5:9:4</td>
<td>MMCRA</td>
<td>yes</td>
<td>64</td>
</tr>
<tr>
<td>86</td>
<td>sp5:9:4</td>
<td>PMC1</td>
<td>yes</td>
<td>32</td>
</tr>
<tr>
<td>87</td>
<td>sp5:9:4</td>
<td>PMC2</td>
<td>yes</td>
<td>32</td>
</tr>
<tr>
<td>88</td>
<td>sp5:9:4</td>
<td>PMC3</td>
<td>yes</td>
<td>32</td>
</tr>
<tr>
<td>89</td>
<td>sp5:9:4</td>
<td>PMC4</td>
<td>yes</td>
<td>32</td>
</tr>
<tr>
<td>91</td>
<td>sp5:9:4</td>
<td>PMC5</td>
<td>yes</td>
<td>32</td>
</tr>
<tr>
<td>92</td>
<td>sp5:9:4</td>
<td>PMC6</td>
<td>yes</td>
<td>32</td>
</tr>
<tr>
<td>95</td>
<td>sp5:9:4</td>
<td>MMCR0</td>
<td>yes</td>
<td>64</td>
</tr>
<tr>
<td>96</td>
<td>sp5:9:4</td>
<td>SIAR</td>
<td>yes</td>
<td>64</td>
</tr>
<tr>
<td>97</td>
<td>sp5:9:4</td>
<td>SDAI</td>
<td>yes</td>
<td>64</td>
</tr>
<tr>
<td>98</td>
<td>sp5:9:4</td>
<td>MMCR1</td>
<td>yes</td>
<td>64</td>
</tr>
<tr>
<td>800</td>
<td>sp5:9:4</td>
<td>BESCRS</td>
<td>no</td>
<td>64</td>
</tr>
<tr>
<td>801</td>
<td>sp5:9:4</td>
<td>BESCRSU</td>
<td>no</td>
<td>64</td>
</tr>
<tr>
<td>802</td>
<td>sp5:9:4</td>
<td>BESCRR</td>
<td>no</td>
<td>64</td>
</tr>
<tr>
<td>803</td>
<td>sp5:9:4</td>
<td>BESCRRU</td>
<td>no</td>
<td>64</td>
</tr>
<tr>
<td>804</td>
<td>sp5:9:4</td>
<td>EBBHR</td>
<td>no</td>
<td>64</td>
</tr>
<tr>
<td>805</td>
<td>sp5:9:4</td>
<td>EBBRR</td>
<td>no</td>
<td>64</td>
</tr>
<tr>
<td>806</td>
<td>sp5:9:4</td>
<td>BESCR</td>
<td>no</td>
<td>64</td>
</tr>
<tr>
<td>808</td>
<td>sp5:9:4</td>
<td>reserved^</td>
<td>no</td>
<td>na</td>
</tr>
</tbody>
</table>
Figure 18. SPR encodings (Sheet 3 of 3)

<table>
<thead>
<tr>
<th>decimal</th>
<th>SPR^†</th>
<th>Register Name</th>
<th>Privileged</th>
<th>Length (bits)</th>
<th>Extended Mnemonics^*</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>sp5:9</td>
<td>sp0:4</td>
<td>mtspr</td>
<td>mfspr</td>
<td>mtspr</td>
</tr>
<tr>
<td>809</td>
<td>11001</td>
<td>01001</td>
<td>reserved^a</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>810</td>
<td>11001</td>
<td>01010</td>
<td>reserved^a</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>811</td>
<td>11001</td>
<td>01011</td>
<td>reserved^a</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>815</td>
<td>11001</td>
<td>01110</td>
<td>TAR</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>816</td>
<td>11001</td>
<td>10000</td>
<td>ASDR</td>
<td>hypv^2</td>
<td>hypv^2</td>
</tr>
<tr>
<td>823</td>
<td>11001</td>
<td>10111</td>
<td>PSSCR</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>848</td>
<td>11010</td>
<td>10000</td>
<td>IC</td>
<td>hypv^2</td>
<td>yes</td>
</tr>
<tr>
<td>849</td>
<td>11010</td>
<td>10001</td>
<td>VTB</td>
<td>hypv^2</td>
<td>yes</td>
</tr>
<tr>
<td>855</td>
<td>11010</td>
<td>10111</td>
<td>PSSCR</td>
<td>hypv^2</td>
<td>hypv^2</td>
</tr>
<tr>
<td>896</td>
<td>11100</td>
<td>00000</td>
<td>PPR</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>898</td>
<td>11100</td>
<td>00010</td>
<td>PPR32</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>1023</td>
<td>11111</td>
<td>11111</td>
<td>PIR</td>
<td>-</td>
<td>yes</td>
</tr>
</tbody>
</table>

^This register is not defined for this instruction.

1. Note that the order of the two 5-bit halves of the SPR number is reversed.
2. This register is a hypervisor resource, and can be accessed by this instruction only in hypervisor state (see Chapter 2).
3. This register cannot be directly written. Instead, bits in the register corresponding to 0 bits in (RS) can be cleared using *mtspr SPR,RS*.
4. 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.
5. The value specified in register RS may be ANDed with the contents of the AMR before being placed into the IAMR; see the *mtspr* instruction description.
6. MMCR0PMCC controls the availability of this SPR, and its contents depend on the privilege state in which it is accessed. See Section 9.4.4 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. Accesses to these SPRs are no-ops; see Section 1.3.3, “Reserved Fields, Reserved Values, and Reserved SPRs” in Book I.
9. The length of the GSR is undefined. An *mtspr* instruction specifying this SPR affects synchronization of subsequent *mtspr* instructions. See the introductory text in this section for more details.
10. SPR numbers 777-778, 783, 793-794, and 799 are reserved for the Performance Monitor. All other SPR numbers that are not shown above and are not implementation-specific are reserved.
11. 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 6.1 of Book II).
12. 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.
13. *mtspr* specifying PIDR performs the implementation specific lookaside invalidation specified by slbia with IH=0b011 when using Radix Tree translation.
14. *mtspr* specifying LPIDR performs the implementation specific lookaside invalidation specified by slbia with IH=0b110 when switching between partitions both of which use Radix Tree translation.

^*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:** *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

mtspr SPR, RS

<table>
<thead>
<tr>
<th>n</th>
<th>spr</th>
<th>31:21</th>
<th>467</th>
<th>/ 31</th>
</tr>
</thead>
<tbody>
<tr>
<td>spr</td>
<td>0</td>
<td>RS</td>
<td>spr</td>
<td>467</td>
</tr>
</tbody>
</table>

n ← spr5g || spr04

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(48): SPR(n) ← (RS)
if PATEHR=1 for the partition then
All implementation-specific lookaside information that was created when address translation was enabled and for which effPID ≠ 0 is invalidated.
case (157): if MSRHV PR = 0b10 then
SPR(157) ← (RS)
else
SPR(157) ← (RS) & AMOR
case (158): start mtspr sequence optimization
case (319): SPR(n) ← (RS)
if PATEHR=1 for both the originating and destination partitions then
All implementation-specific lookaside information that was created when address translation was enabled and for which effPID ≠ 0 is invalidated.
case (336): SPR(336) ← (SPR(336)) & (RS)
case (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 18. If the SPR field contains the value 158, the instruction indicates the start of a sequence of mtspr instructions that may be synchronized as a group. See the introductory material in this section for more information. 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 four 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 UAMOR and MSRHV PR=0b00, the contents of register RS are ANDed with the contents of the AMOR and the result is placed into the UAMOR.

When the designated SPR is the PIDR and the partition uses Radix Tree translation, the implementation specific lookaside invalidation specified by slbia with IH=0b011 is performed along with the SPR update. When the designated SPR is the LPIDR and both the originating and destination partitions use Radix Tree translation, the implementation specific lookaside invalidation specified by slbia with IH=0b110 is performed along with the SPR update.

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 causes a Privileged Instruction type Program interrupt when MSRPR=1 and, if the SPR is a hypervisor resource (see Figure 18) when MSRHV PR=0b00, causes a Privileged Instruction type Program interrupt if LPCREVR=0 and a Hypervisor Emulation Assistance interrupt if LPCREVR=1.

Execution of this instruction specifying an SPR number that is undefined for the implementation causes one of the following.

- if spr0=0:

974 Power ISA™ III
- if MSRPR=1: Hypervisor Emulation Assistance interrupt
- if MSRPR=0: Hypervisor Emulation Assistance interrupt for SPR 0, 4, 5, and 6, and 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

- 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

If an attempt is made to execute mtspr specifying a Transactional Memory SPR in other than Non-transactional state, with the exception of TFIAR in suspended state, a TM Bad Thing type Program interrupt is generated.

Special Registers Altered:
See Figure 18

--- Programming Note ---
For a discussion of software synchronization requirements when altering certain Special Purpose Registers, see Chapter 11. “Synchronization Requirements for Context Alterations” on page 1133.

--- 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 mtspr 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.)

The SPR field denotes a Special Purpose Register, encoded as shown in Figure 18. If the designated Special Purpose Register is the TFIAR and TFIAR indicates the failure was recorded in a state more privileged than the current state, register RT is set to zero. 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.

--- Programming Note ---
Note that when a problem state transaction’s failure is recorded in hypervisor state and there is a subsequent need for a context switch in privileged, non-hypervisor state, an attempt to save TFIAR will result in zeros being saved. This is harmless because if the original application ever tries to read the TFIAR, it would read zeros anyway, since the failure took place in hypervisor state.

--- Programming Note ---
spr0=1 if and only if reading the register is privileged. Execution of this instruction specifying an SPR number with spr0=1 causes a Privileged Instruction type Program interrupt when MSRPR=1 and, if the SPR is a hypervisor resource (see Figure 18) when MSRHVPR=0b00, causes a Privileged Instruction type Program interrupt when LPCREVIRT=0 and a Hypervisor Emulation Assistance interrupt when LPCREVIRT=1.
Execution of this instruction specifying an SPR number that is not defined for the implementation causes one of the following.

- if spr0=0:
  - if MSRPR=1: Hypervisor Emulation Assistance interrupt
  - 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 LPCREVIRT=0 and Hypervisor Emulation Assistance interrupt when LPCREVIRT=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 LPCREVIRT=0 and Hypervisor Emulation Assistance interrupt when LPCREVIRT=1

Special Registers Altered:
None

Note
See the Notes that appear with mtspr.
Move To Machine State Register X-form

\[ \text{mtmsr} \quad RS, L \]

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>5</th>
<th>11</th>
<th>15</th>
<th>16</th>
<th>21</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>RS</td>
<td>L</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

if L = 0 then
\[
\text{MSR}_{48} \leftarrow (RS)_{48} \quad (RS)_{49}
\]
\[
\text{MSR}_{58} \leftarrow (RS)_{58} \quad (RS)_{49}
\]
\[
\text{MSR}_{59} \leftarrow (RS)_{59} \quad (RS)_{49}
\]
\[
\]

else
\[
\text{MSR}_{48} \quad 62 \leftarrow (RS)_{48} \quad 62
\]

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 ORing bits 58 and 49 of register RS is placed into MSR\(_{58}\). The result of ORing bits 59 and 49 of register RS is placed into MSR\(_{59}\). Bits 32:47, 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 \text{mtmsr} instruction description in this section, references to “\text{mtmsr}” in this document imply either L value unless otherwise stated or obvious from context (e.g., a reference to an \text{mtmsr} instruction that modifies an MSR bit other than the EE or RI bit implies L=0).

\[ \text{Programming Note} \]

If this instruction sets MSR\(_{PR}\) to 1, it also sets MSR\(_{EE}\), MSR\(_{IR}\), and MSR\(_{OD}\) to 1.

This instruction does not alter MSR\(_{ME}\) or MSR\(_{LE}\). (This instruction does not alter MSR\(_{HV}\) because it does not alter any of the high-order 32 bits of the MSR.)

If the only MSR bits to be altered are MSR\(_{EE}\) \& RI, to obtain the best performance L=1 should be used.

\[ \text{Programming Note} \]

If MSR\(_{EE}=0\) and an External, Decrementer, or Performance Monitor exception is pending, executing an \text{mtmsrd} instruction that sets MSR\(_{EE}\) to 1 will cause the interrupt to occur before the next instruction is executed, if no higher priority exception exists (see Section 6.9, “Interrupt Priorities” on page 1092). 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 11.

\[ \text{Programming Note} \]

\text{mtmsr} serves as both a basic and an extended mnemonic. The Assembler will recognize an \text{mtmsr} mnemonic with two operands as the basic form, and an \text{mtmsr} mnemonic with one operand as the extended form. In the extended form the L operand is omitted and assumed to be 0.

\[ \text{Programming Note} \]

There is no need for an analogous version of the \text{mtmsrd} instruction, because the existing instruction copies the entire contents of the MSR to the selected GPR.
**Move To Machine State Register Doubleword**  

**X-form**

\[
\text{mtmsrd} \quad \text{RS}, L
\]

<table>
<thead>
<tr>
<th>31</th>
<th>15</th>
<th>16</th>
<th>21</th>
<th>178</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

if L = 0 then

if (MSR\(_{29:31}\) ≠ 0b010 | RS\(_{29:31}\) ≠ 0b000) then

\[
\begin{align*}
\text{MSR}_{48} & \leftarrow (\text{RS})_{48} \\
\text{MSR}_{58} & \leftarrow (\text{RS})_{58} \\
\text{MSR}_{59} & \leftarrow (\text{RS})_{59} \\
\text{MSR}_{0:2} & \leftarrow 4; 28; 32; 47; 49; 50; 52; 57; 60; 62 \\
\end{align*}
\]

else

\[
\begin{align*}
\text{MSR}_{48} & \leftarrow (\text{RS})_{48} \\
\text{MSR}_{62} & \leftarrow (\text{RS})_{62}
\end{align*}
\]

The MSR is set based on the contents of register RS and of the L field.

L=0:

If bits 29 through 31 of the MSR are not equal to 0b010 or bits 29 through 31 of RS are not equal to 0b000, then the value of bits 29 through 31 of RS is placed into bits 29 through 31 of the MSR. The result of ORing bits 48 and 49 of register RS is placed into MSR\(_{48}\). The result of ORing bits 58 and 49 of register RS is placed into MSR\(_{58}\). The result of ORing bits 59 and 49 of register RS is placed into MSR\(_{59}\). Bits 0, 2, 4, 28, 32, 47, 49, 50, 52, 57, 60, 62 of register RS are placed into the corresponding bits of the MSR.

L=1:

If the instruction attempts to cause an illegal transaction state transition or, when TM is made unavailable in problem state by the PCR, attempts to cause a transition to problem state and also a transaction state transition that Table 3 on page 947 shows as legal and as resulting in the thread being in Transactional or Suspended state, a TM Bad Thing type Program interrupt is generated (unless a higher-priority exception is pending). If this interrupt is generated, the value placed into SRR0 by the interrupt processing mechanism (see Section 6.4.3) is the address of the mtmsrd instruction. This instruction is privileged.

This instruction is context synchronizing. 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.

This instruction does not alter MSR\(_{LE}\), MSR\(_{ME}\), or MSR\(_{HV}\).

If the only MSR bits to be altered are MSR\(_{EE}\), MSR\(_{IR}\), to obtain the best performance L=1 should be used.

---

**Programming Note**

If MSR\(_{EE}\)=0 and an External, Decrementer, or Performance Monitor exception is pending, executing an mtmsrd instruction that sets MSR\(_{EE}\) to 1 will cause the interrupt to occur before the next instruction is executed, if no higher priority exception exists (see Section 6.9, “Interrupt Priorities” on page 1092). 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 11.

---

**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

```
mfmsr   RT
```

RT ← MSR

The contents of the MSR are placed into register RT.

This instruction is privileged.

Special Registers Altered:

None
Chapter 5. Storage Control

5.1 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 5.7.3, in Section 5.7.7 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 1.11 of Book I.

5.2 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 2.2 of Book II, and Section 6.6 in this Book.

5.3 Instruction Fetch

Instructions are fetched under control of MSRIR:

\[ \text{MSR}_\text{IR}=0 \]

The effective address of the instruction is interpreted as described in Section 5.7.3.

\[ \text{MSR}_\text{IR}=1 \]

The effective address of the instruction is translated by the Address Translation mechanism described beginning in Section 5.7.7.

5.3.1 Implicit Branch

Explicitly altering certain MSR bits (using \textit{mtmsr[0]}), or explicitly altering SLB entries, Page Table Entries, or certain System Registers (including the HRMOR, 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 \textit{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 11. “Synchronization Requirements for Context Alterations” on page 1133. Implicit branches are not supported by the Power ISA. If an implicit branch occurs, the results are boundedly undefined.

5.3.2 Address Wrapping Combined with Changing MSR Bit SF

If the current instruction is at effective address \(2^{32} - 4\) and is an \textit{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 at effective address \(2^{32} - 4\), 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.
5.4 Data Access

Data accesses are controlled by MSR_{DR}.

**MSR_{DR}=0**

The effective address of the data is interpreted as described in Section 5.7.3.

**MSR_{DR}=1**

The effective address of the data is translated by the Address Translation mechanism described in Section 5.7.7.

5.5 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 Branch, Trap, System Call, and 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 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 Load or Store instruction (e.g., a Load or 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.

- Stores are not performed out-of-order (even if the Store instructions that caused them were executed out-of-order).
- Accessing Guarded Storage
  The restrictions for this case are given in Section 5.8.1.1.
  The only permitted side effects of performing an operation out-of-order are the following.
  - A Machine Check or Checkstop that could be caused by in-order execution may occur out-of-order.
  - Reference and Change bits may be set as described in Section 5.7.12.
  - 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.
  - SPRs that are specified by mtsp {r} instructions within a sequence initiated by mtgsr (see Chapter 11 of Book III) may be modified in arbitrary relative order, except that if an interrupt occurs within the sequence, all mtsp {r} instructions in the sequence prior to the point of interruption appear to have been executed before the first instruction of the interrupt handler is executed.

5.6 Invalid Real Address

A storage access (including an access that is performed out-of-order; see Section 5.5) 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 6.5.2 on page 1067.

---

**Programming Note**

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.
5.7 Storage Addressing

Storage Control Overview

- Host real address space size is $2^m$ bytes, $m \leq 60$; see Note 1.
- Guest real address space size is $2^m$ bytes, $m \leq 60$; 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, $65 \leq n \leq 78$; see Note 3.
  - Segment size is $2^s$ bytes, $s=28$ or 40.
  - $2^{n-40} \leq$ number of virtual segments $\leq 2^{n-28}$; see Note 3.
  - Virtual page size is $2^p$ bytes, where $12 \leq p$, 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 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.
  - 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 $12 \leq p$, and $2^p$ is no larger than the size of the real address space; a size of 4 KB, 64 KB, 2 MB, and an implementation-dependent number of other sizes are supported; see Note 4. The virtual page size is determined by the location of the Page Table Entry in the Radix Tree.

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.

5.7.1 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 (MSR$_{SF}=0$), 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.
5.7.2 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 5.7.7 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 5.7.3.

5.7.3 Hypervisor Real And Virtual Real Addressing Modes

When a storage access is an instruction fetch performed when instruction address translation is disabled, or a data access performed when data address translation is disabled, it is said to be performed in “hypervisor real addressing mode” if the thread is in hypervisor state. If the thread is not in hypervisor state, the access is said to be performed in “virtual real addressing mode.” Storage accesses in hypervisor real and virtual real addressing modes are performed in a manner that depends on the contents of MSRVR, PATER, PATEPS, HRMOR (see Chapter 2), bit 0 of the effective address (EA0), and the state of the Real Mode Storage Control Facility as described below.

Bits 1:3 of the effective address are ignored.

MSRVR=1
- If EA0=0, the Hypervisor Offset Real Mode Address mechanism, described in Section 5.7.3.1, controls the access.
- If EA0=1, bits 4:63 of the effective address are used as the real address for the access.

MSRVR=0
- If PATER=0, the Virtual Real Mode Addressing mechanism, described in Section 5.7.3.3, controls the access.
- If PATER=1, partition-scoped translation is performed on the effective address. (See Section 5.7.11.3, “Obtaining Host Real Address, Radix on Radix”.)

5.7.3.1 Hypervisor Offset Real Mode Address

If MSRVR = 1 and EA0 = 0, the access is controlled by the contents of the Hypervisor Real Mode Offset Register, as follows.

Hypervisor Real Mode Offset Register (HRMOR)

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. The supported offset values are all values of the form \( i \times 2^r \), 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.

Programming Note

If \( m < 60 \), \( EA_{4:63-m} \) and \( HRMOR_{4:63-m} \) must be zeros.

5.7.3.2 Storage Control Attributes for Accesses in Hypervisor Real Addressing Mode

Storage accesses in hypervisor real addressing mode 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 5.7.3.2.1). (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
Additionally, storage accesses in hypervisor real addressing mode are performed as though all storage was not No-execute.

**Programming Note**

Because storage accesses in hypervisor real addressing mode do not use the SLB or the Page Table, accesses in this mode 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).

### 5.7.3.2.1 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 non-Guarded in hypervisor real addressing mode (MSRHV=0b10, and MSRIR=0 or MSRDR=0, as appropriate for the type of access). The remaining portions are treated as Guarded in hypervisor real addressing mode. 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 non-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 non-Guarded for subsequent accesses on a best effort basis, limited by the amount of history that the facility can maintain.

The storage location specified by a Load/Store Caching Inhibited instruction must not be in storage that is specified by the Hypervisor Real Mode Storage Control facility to be treated as non-Guarded. The storage location specified by any other Load or Store instruction must not be in storage that is specified by the Hypervisor Real Mode Storage Control facility to be treated as Guarded. ("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 5.9.3.2.

**Programming Note**

There are two cautions about mixing different types of accesses (i.e., Load/Store Caching Inhibited instructions vs. any other Load or Store instruction vs. instruction fetches). The first, as indicated above, is to avoid confusing the history mechanism, and the granularity for concern is a history block. For this caution, instruction fetches are irrelevant because they have their own history mechanism and are always intended to be non-guarded.

The second caution is to avoid storage paradoxes that result from a Caching Inhibited access to a location that is held in a cache. The nature of this caution and its solution are described in Section 5.8.2.2, "Altering the Storage Control Bits". The minimum granularity for concern is the history block, but may be larger, depending on extant translations to the storage in question. Since the consistency of instruction storage is managed by software and hypervisor real mode instruction fetches are always not Caching Inhibited, instruction fetches are also irrelevant to this caution.

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 5.7.3.4.

**Programming Note**

The preceding capability can be used to improve the performance of hypervisor software that runs in hypervisor real addressing mode, by causing accesses to instructions and data that occupy well-behaved storage to be treated as non-Guarded.

### 5.7.3.3 Virtual Real Mode Addressing Mechanism

If MSRHV=0, the partition is using Paravirtualized HPT translation (PATEHR=0), and MSRDR=0 or MSRIR=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 19 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.

**Programming Note**

The Virtual Page Class Key Protection mechanism does not apply because the authority mask that an OS has set for application programs executing with address translation enabled may not be the same as the authority mask required by the OS when address translation is disabled, such as when first entering an interrupt handler.

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>(K_s)</td>
<td>0</td>
</tr>
<tr>
<td>(K_p)</td>
<td>undefined</td>
</tr>
<tr>
<td>(N)</td>
<td>0</td>
</tr>
<tr>
<td>(L)</td>
<td>PATE(_{PS}[0])</td>
</tr>
<tr>
<td>(C)</td>
<td>0</td>
</tr>
<tr>
<td>(LP)</td>
<td>PATE(_{PS}[1:2])</td>
</tr>
</tbody>
</table>

**Figure 19. SLBE for VRMA**

**Programming Note**

The \(C\) bit in Figure 19 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 5.7.8.1.

**Programming Note**

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 \(PTE_B = 0b01\) for all Page Table Entries that map the VRMA in order to be consistent with the values in Figure 19.

### 5.7.3.4 Storage Control Attributes for Implicit Storage Accesses

Implicit accesses to the Partition Table and to a partition-scoped Page Table during address translation and in recording reference and change information are performed as though the storage occupied by the tables had the following storage control attributes.

- not Write Through Required
- not Caching Inhibited
- Memory Coherence Required
- not Guarded
- not SAO

Implicit accesses to a Process Table, Segment table, or process-scoped Page Table during address translation and in recording reference and change information are performed using the storage control attributes in the partition-scoped Page Table Entry that maps the other In-Memory Table Entry or the process-scoped Page Table Entry that is being accessed. The storage control attributes must be those described above.

### 5.7.4 Definitions

- **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 MSR\(_{HV}=1\) , or Segment translation.
- **partition-scoped**: Refers to translation performed using table(s) 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 MSR\(_{HV}=1\) . 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 “off” 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 5.7.5.1, “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 MSRPR=0b11, thereby restricting the resources it can modify (MSRPR=1) and causing its interrupts to go to the hypervisor (MSRHV=1). 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, LPID/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 a hypervisor that uses HPT translation and only for LPIDR=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.

5.7.5 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 page beginning at the real address specified by the HRMOR is used similarly to the page for the fixed interrupt vectors.

- Relocated interrupt vectors

  Depending on the values of MSRIRDR and LPCRAIL and on whether the specific interrupt will cause MSRHV to change, either the virtual page containing the byte addressed by effective address 0x0000_0000_0001_8000 or 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 value of LPCRAIL, the virtual page containing the effective address 0x0000_0000_0001_7000 or 0xC000_0000_0000_3000 contains the interrupt vectors that are invoked by the System Call Vectored instruction.

- 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.

5.7.5.1 Effective Address Space Structure for Radix-using Partitions

When Radix Tree translation is in use but translation is disabled (MSRR=0), MSRHV selects between partition-scoped translation of the real mode guest real address, formed by treating EA0:1 as 0b00, and hypervisor real mode (see Section 5.7.3). When Radix Tree translation is in use and translation is enabled, EA0:1 together with MSRHV are used to select one of as many as four Radix Trees with 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 20 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. 32b software will only be able to access storage mapped by its own Radix Tree.

---

**Programming Note**

**Warning:** The functionality described in this section, e.g. directing most hypervisor interrupts to the LPID=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 and the guest operating system, quadrant 0 (EA0:1=0b00) addresses the Radix Tree for the application and quadrant 3 (EA0:1=0b11) addresses the direct supervisor of the application. For the guest and host applications, it will frequently be the case that page protection is used to prevent access to quadrant 3, but partition-wide shared text and/or data may also be located there. Quadrants 1 and 2 have no associated Radix Tree.

**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, quadrants 0 and 3 are as described above. Quadrant 1 (EA0:1=0b01) addresses the guest application and quadrant 2 (EA0:1=0b10) addresses the guest operating system, one of which experienced a hypervisor interrupt or performed a system call to the hypervisor. It will rarely be the case that quadrants 0 and 1 will be in use concurrently. A new value will usually be put in PIDR between accesses to quadrants 0 and 1.

When MSR$_{HV}$=1 and EA$_{0:1}$=0b00 or 0b11, only process-scoped translation is performed. When MSR$_{HV}$=0 and MSR$_{R/DR}$=0, only partition-scoped translation is performed. Otherwise, nested process- and partition-scoped translations are performed.

---

**Figure 20. Effective address space structure when using Radix Tree translation**

<table>
<thead>
<tr>
<th>Guest</th>
<th>Host App</th>
<th>Hypervisor</th>
</tr>
</thead>
<tbody>
<tr>
<td>EA$_{0:1}$=0b11</td>
<td>effPID=0</td>
<td>effPID=0</td>
</tr>
<tr>
<td>effLPID=LPIDR</td>
<td>effLPID=LPIDR</td>
<td>effLPID=LPIDR</td>
</tr>
<tr>
<td>EA$_{0:1}$=0b00</td>
<td>effPID=PIDR</td>
<td>effPID=PIDR</td>
</tr>
<tr>
<td>effLPID=LPIDR</td>
<td>effLPID=LPIDR</td>
<td>effLPID=LPIDR</td>
</tr>
</tbody>
</table>

---

**5.7.6 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 `tlbMem[fi]` 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 (PRTEP TPS, see Figure 23) using LPCR ISL. The base page size for the Process Table (PAT EPRTP S) 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.

#### 5.7.6.1 Partition Table

The Partition Table Control Register (PTCR) is a hypervisor privileged SPR 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>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+\text{PAT S})</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(\text{PAT S} \leq 24)</td>
</tr>
</tbody>
</table>

All other fields are reserved.

**Example:**

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 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 30) 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 \(n-39\). Because the high-order \(78-n\) bits of the VSID are assumed to be zeros, the hash value used in the Page Table search will have the high-order \(67-n\) bits either all 0s (primary hash; see Section 5.7.9.2) or all 1s (secondary hash). If HTABSIZE > \(n-39\), some of these hash value bits will be used to index into the Page Table, with the result that certain PTEGs will not be searched.

**Programming Note**

Let \(n\) equal the virtual address size (in bits) supported by the implementation. If \(n=67\), software should set the HTABSIZE field to a value that does not exceed \(n-39\). Because the high-order 78-\(n\) bits of the VSID are assumed to be zeros, the hash value used in the Page Table search will have the high-order 67-\(n\) bits either all 0s (primary hash; see Section 5.7.9.2) or all 1s (secondary hash). If HTABSIZE > \(n-39\), some of these hash value bits will be used to index into the Page Table, with the result that certain PTEGs will not be searched.
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.

<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>HR</td>
<td>Host Radix</td>
</tr>
<tr>
<td>0b0-</td>
<td>hypervisor uses HPT translation for this partition</td>
<td></td>
</tr>
<tr>
<td>0b1-</td>
<td>hypervisor uses Radix Tree translation for this partition</td>
<td></td>
</tr>
<tr>
<td>4:45</td>
<td>HTABORG</td>
<td>Hashed Page Table Base</td>
</tr>
<tr>
<td>56:58</td>
<td>PS</td>
<td>Page Size (uses L</td>
</tr>
<tr>
<td>59:63</td>
<td>HTABSIZE</td>
<td>HPT size = 2^{HTABSIZE+18}</td>
</tr>
<tr>
<td>1:38</td>
<td>PRTB</td>
<td>Process Table Base (when UPRT=1)</td>
</tr>
<tr>
<td>56:58</td>
<td>PRTPS</td>
<td>Process Table Page Size (when UPRT=1) (uses L</td>
</tr>
<tr>
<td>59:63</td>
<td>PRTS</td>
<td>Process Table Size = 2^{12+PRTS} PRTS≤24 (when UPRT=1)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>RTS1</td>
<td>Radix Tree Size[0:1]</td>
</tr>
<tr>
<td>1:2</td>
<td>RPDB</td>
<td>Root Page Directory Base</td>
</tr>
<tr>
<td>4:55</td>
<td>RTS2</td>
<td>Radix Tree Size[2:4] (number of address bits mapped), size=2^{RTS+31}</td>
</tr>
<tr>
<td>59:63</td>
<td>RPDS</td>
<td>Root Page Directory Size = 2^{RPDS+3}, RPDS≥5</td>
</tr>
<tr>
<td>4:51</td>
<td>PRTB</td>
<td>Process Table Base</td>
</tr>
<tr>
<td>59:63</td>
<td>PRTS</td>
<td>Process Table Size = 2^{12+PRTS} PRTS≤24 (when UPRT=1)</td>
</tr>
</tbody>
</table>

All other fields are reserved.

**Figure 22. 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 22. 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.

5.7.6.2 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).

Programming Note

The size of the Process Table is provided to simplify hardware design and testing. The size enables the hardware to mask address bits instead of providing an adder. No size checking is provided for these tables. An out-of-range LPID or PID will not produce an exception simply because of its size. Hypervisor software may help detect such errors by the OS by not providing a translation for virtual / guest real addresses for a page or two beyond the end of the Process Table.

5.7.7 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 5.2) occurs.

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 MSRHV=1 or MSRDR=0 (as appropriate for the type of access), the steps taken for a given mode vary. See Sections 5.7.11.3 and 5.7.11.4 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 5.7.8, and HPT translation to the virtual address to produce a host real address as described in Section 5.7.9. 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 24 gives an overview of the address translation process.

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 25 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 36 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 overcommitment. 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 24. Address translation overview](image-url)
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. 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 5.10).

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. In this case, when a valid TLB entry is created, the LPID value from LPIDR is written into the TLB entry.

**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 virtual page.
5.7.8 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 26. If no matching translation is found in the SLB, LPCR\textsubscript{UPRT}=1, and either MSR\textsubscript{AV}=0 or LPID=0, the Segment Table is searched. For implicit accesses, implicit segment descriptors are provided, as described elsewhere in this chapter.

Figure 26. Translation of 64-bit effective address to 78 bit virtual address

5.7.8.1 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\textsubscript{UPRT}=0 all of the entries, of the SLB are managed by software, using the instructions described in Section 5.9.3.2. See Chapter 11. “Synchronization Requirements for Context Alterations” on page 1133 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 27 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\textsubscript{s}</td>
<td>Supervisor (privileged) state storage key (see Section 5.7.13.2)</td>
</tr>
<tr>
<td>90</td>
<td>K\textsubscript{p}</td>
<td>Problem state storage key (See Section 5.7.13.2.)</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\textsubscript{0} (SLBE\textsubscript{37}) is treated as a reserved field.

Figure 27. 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 SLBL\textsubscript{LILP} encodings are those shown in Figure 28. 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\textsuperscript{b} bytes. The actual virtual page size (also referred to as the “actual page size” or “virtual page size”) is specified by PTE\textsubscript{LILP}. 

Figure 28. SLB Entry
### Figure 28. Page Size Encodings

For each SLB entry, software must ensure the following requirements are satisfied.

- \( L||LP \) contains a value supported by the implementation.
- The base virtual page size selected by the \( L \) and \( LP \) fields does not exceed the segment size selected by the \( B \) field.
- If \( s=40 \), the following bits of the SLB entry contain 0s.
  - \( \text{ESID}_{24:35} \)
  - \( \text{VSID}_{38:49} \)

The bits in the above two items are ignored by the hardware.

<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**

- 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.

**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 11 Note 10.

### 5.7.8.2 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.

- \( V=1 \)
- \( \text{ESID}_{63-s} = \text{EA}_{63-s} \), 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.

\[
VA = \text{VSID}_{77-s} || \text{EA}_{64-s:63}
\]

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 5.7.9.1). If \( \text{SLBE}_N = 1 \), 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.

### 5.7.8.3 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 \( q = \text{STAB}-\text{SIZE}+12, \log_2(\text{size of the Segment Table}) \). The base of the Segment Table in virtual address space is \( \text{STABORG}_{0:77-q} || 0^{q-12} \). Software must set the low 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.

<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 - 256 MB (s=28)</td>
<td>0b01 - 1 TB (s=40)</td>
</tr>
<tr>
<td></td>
<td>0b10 - reserved</td>
<td>0b11 - 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</td>
</tr>
<tr>
<td>117</td>
<td>K_p</td>
<td>Problem state storage key</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 29. Segment Table Entry

#### 5.7.8.3.1 Primary Hash for 256MB Segment

The STEG is located at host VA

\[
\text{STABORG}_{0:77-q} \ || \ EA_{43:35} \ || \ 0b0000000.
\]

Each of the 8 STEs are searched to find a valid entry \((V=1, B=0b00)\) that matches the ESID \((\text{STEESID}[0:35] = EA_{0:35})\) of the access being translated.

#### 5.7.8.3.2 Primary Hash for 1TB Segment

The STEG is located at host VA

\[
\text{STABORG}_{0:77-q} \ || \ EA_{31-q:23} \ || \ 0b0000000.
\]

Each of the 8 STEs are searched to find a valid entry \((V=1, B=0b01)\) that matches the ESID \((\text{STEESID}[0:23] = EA_{0:23})\) of the access being translated.

#### 5.7.8.3.3 Secondary Hash for 256MB Segment

The STEG is located at host VA

\[
\text{STABORG}_{0:77-q} \ || \ \neg EA_{43:35} \ || \ 0b0000000.
\]

Each of the 8 STEs are searched to find a valid entry \((V=1, B=0b00)\) that matches the ESID \((\text{STEESID}[0:35] = EA_{0:35})\) of the access being translated.

#### 5.7.8.3.4 Secondary Hash for 1TB Segment

The STEG is located at host VA

\[
\text{STABORG}_{0:77-q} \ || \ \neg EA_{31-q:23} \ || \ 0b0000000.
\]

Each of the 8 STEs are searched to find a valid entry \((V=1, B=0b01)\) that matches the ESID \((\text{STEESID}[0:23] = EA_{0:23})\) of the access being translated.

---

### 5.7.9 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 30.
Figure 30. Translation of 78-bit virtual address to 60-bit real address
5.7.9.1 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 5.7.3.4). 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 5.10 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 31 shows the layout of a PTE. This layout is independent of the Endian mode of the thread.

<table>
<thead>
<tr>
<th>Dword</th>
<th>Bit(s)</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>12:56</td>
<td>AVA</td>
<td>Abbreviated Virtual Address</td>
</tr>
<tr>
<td>57:60</td>
<td>SW</td>
<td>Available for software use</td>
<td></td>
</tr>
<tr>
<td>61</td>
<td>L</td>
<td>Virtual page size</td>
<td>0b0 - 4 KB</td>
</tr>
<tr>
<td>62</td>
<td>H</td>
<td>Hash function identifier</td>
<td></td>
</tr>
<tr>
<td>63</td>
<td>V</td>
<td>Entry valid (V=1) or invalid (V=0)</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>pp</td>
<td>Page Protection bit 0</td>
<td></td>
</tr>
<tr>
<td>2:3</td>
<td>key</td>
<td>KEY bits 0:1</td>
<td></td>
</tr>
<tr>
<td>4:5</td>
<td>B</td>
<td>Segment Size</td>
<td>0b00 - 256 MB</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0b01 - 1 TB</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0b10 - reserved</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0b11 - reserved</td>
</tr>
</tbody>
</table>

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 $b\leq23$, 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.

### Programming Note

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 and LP 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=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.
If PTEL=1, the base page size and actual page size are specified by PTELP. In this case, the contents of PTELP have the format shown in Figure 32. 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 “z” 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 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.

\[
\begin{array}{c|c}
\text{PTE LP} & \text{actual page size} \\
\hline
rrrr \_ rrrz & \geq 8 \text{ KB} \\
rrrr \_ rrrzz & \geq 16 \text{ KB} \\
rrrr \_ rzzzz & \geq 32 \text{ KB} \\
rrrr \_ zzzzz & \geq 64 \text{ KB} \\
rzzz \_ zzzzz & \geq 128 \text{ KB} \\
rzzz \_ zzzzz & \geq 256 \text{ KB} \\
rzzz \_ zzzzz & \geq 512 \text{ KB} \\
zzzz \_ zzzzz & \geq 1 \text{ MB}
\end{array}
\]

Figure 32. Format of PTELP when PTEL=1

There are at least 2 formats of PTELP that specify a 64 KB page. One format is used with SLBEL||LP = 0b000 and one format is used with SLBEL||LP = 0b101.

The actual page size selected by the LP field must not exceed the segment size selected by the B field. Forms of PTELP not supported by a given implementation are treated as reserved values for that implementation.

The concatenation of the ARPN field and bits labeled “r” in the LP field contain the high-order bits of the real page number of the real page that maps the first 4KB of the virtual page described by the entry. The low-order p-12 bits of the real page number contained in the ARPN and LP fields must be 0s and are ignored by the hardware.

Programming Note

The actual page size specified by a given PTELP format is at least \(2^{12+8-c}\), where \(c\) is the number of \(r\) bits in the format.

![Programming Note](image)

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 5.7.6.1) 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 \(2^{18}\) PTEGs, or \(2^{25}\) bytes (32 MB).

5.7.9.2 Page Table Search

When the hardware searches the Page Table, the accesses are performed as described in Section 5.7.3.4.

An outline of the HTAB search process is shown in Figure 30. 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 \( (11^{t=0})||VA_{50:77-b} \)

If \( s=40 \), the hash value is computed by Exclusive ORing the following three quantities: \( VA_{24:37}||250 \), \( (0)||VA_{0:37} \), 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.

2. **Secondary Hash:**

If the secondary Page Table search is enabled (\( LPCRTC=0 \)), 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 \( (11^{t=0})||VA_{50:77-b} \)

If \( s=40 \), the hash value is computed by taking the ones complement of the Exclusive OR of the following three quantities: \( VA_{24:37}||250 \), \( (0)||VA_{0:37} \), 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\(_{V}=1\)
- PTE\(_{B}=SLBE_{B}\)
- PTE\(_{AVA}[0:q-10]=VA_{10:q}\)
- if \( b=12 \) then 
  \[ \text{PTE}_{L}=0 \] \( \land \) (PTE\(_{LP}\) specifies the 4KB base page size)
  else 
  \[ \text{PTE}_{L}=1 \] \( \land \) (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 \( p = \log_{2}(\text{actual page size specified by PTE}_{L||LP}) \).

- 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.
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.

5.7.10 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 22 and 23.) The Root Page Directory Size (RPDS) is specified as \( \log_2(\text{number of entries in the table}) \). 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 each point to another page of entries, and give its size in the Next Level Size field, PDENLS. The next 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. For an actual page size that is larger than \( 2^{23} \) (8 MB), the PTEAVA differs 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 PTEH 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.

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 \( \log_2(\text{segment's base page size}) \).
- 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 (LPCRTEC=0).

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 33. Four level Radix Tree walk translating a 52b EA with NLS=13 in the root PDE and NLS=9 in the other PDEs.

5.7.10.1 Radix Tree Page Directory Entry

<table>
<thead>
<tr>
<th>V L</th>
<th>/</th>
<th>NLB</th>
<th>///</th>
<th>NLS</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>3</td>
<td>55</td>
<td>58</td>
</tr>
</tbody>
</table>

Root Page Directory Base

Effective Page Number
### 5.7.10.2 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≥5)</td>
</tr>
</tbody>
</table>

All other fields are reserved.

**Figure 34. Radix Tree Page Directory Entry**

### 5.7.10.3 Nested Translation

When MSR\(_{MV}=0\) and translation is enabled, each guest real address must undergo partition-scoped translation using the hypervisor’s Radix Tree for the partition. See Figure 36.

<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>2</td>
<td>sw</td>
<td>SW bit 0</td>
</tr>
<tr>
<td>7:51</td>
<td>RPN</td>
<td>Real Page Number</td>
</tr>
<tr>
<td>52:54</td>
<td>sw</td>
<td>SW bits 1:3</td>
</tr>
<tr>
<td>55</td>
<td>R</td>
<td>Reference</td>
</tr>
<tr>
<td>56</td>
<td>C</td>
<td>Change</td>
</tr>
<tr>
<td>58:59</td>
<td>Att</td>
<td>Attributes (equivalent WIMG value)</td>
</tr>
<tr>
<td>60:63</td>
<td>EAA</td>
<td>Encoded Access Authority</td>
</tr>
</tbody>
</table>

All other fields are reserved.

**Figure 35. Radix Tree Page Table Entry**
Figure 36. Radix on Radix Page Table search for a 52-bit EA depicting memory reads 1-24 numbered in sequence

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). For storage ordering, the SAO attribute takes effect when both SAO and normal memory attributes are specified. (The hypervisor will typically specify "normal memory" and the OS may override that with SAO.)
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 partition-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 4. Together these rules can produce the WIMG=0b00011 state that any individual Att value cannot express.

<table>
<thead>
<tr>
<th>process-scoped Att</th>
<th>partition-scoped Att</th>
<th>00</th>
<th>01</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAO/I/G</td>
<td>00</td>
<td>000</td>
<td>010</td>
<td>011</td>
<td>010</td>
</tr>
<tr>
<td></td>
<td>01</td>
<td>100</td>
<td>100</td>
<td>011</td>
<td>010</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>011</td>
<td>001</td>
<td>011</td>
<td>011</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>010</td>
<td>000</td>
<td>010</td>
<td>010</td>
</tr>
</tbody>
</table>

Table 4: Effective SAO, I and G attributes for nested translation

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 5.7.12, “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 \textit{tlbie} instructions that software will use to manage the logical model.)

5.7.11 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.

5.7.11.1 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:

- \texttt{effLPID}
- \texttt{effPID}
- \texttt{EA}

The additional context required to perform a translation or match a cached translation may include the following:

- \texttt{PATEHR} (selected using the value in \texttt{LPIIDR}, not \texttt{effLPID})
- \texttt{MSRHV PR IR DR}
At a high level, 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} determine where and how the process is entered.

5.7.11.2 Finding the Page Tables

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 translation 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) 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. Some types of translation use process-scoped Page Tables, so use partition-scoped Page Tables, and some use both.

5.7.11.3 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 if LPIDR≠0.
- 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 if LPIDR≠0.
- Guest OS access with translation off: partition-scoped translation is performed on LPIDR||PIDR||EA, with the result subject to partition-scoped translation 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 real mode access: subject to HRMOR and EA_{0} as described in Section 5.7.3.1.

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 table, but it may be set up to point to a table used solely to translate the addresses of Process Table Entries.
5.7.11.4 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 when MSRHv=1, the LPID value is always taken from LPIDR and the PID value is always taken from PIDR. 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 5.7.11.2, “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 5.7.8. 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 5.7.3.3.
- Hypervisor real mode access: subject to HRMOR and EA0 as described in Section 5.7.3.1.

5.7.12 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, the Reference and Change bits are set 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.) For HPT translation, Reference and Change bits are set 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. 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. If hardware is unable to set the bit(s) atomically for Radix Tree translation, a [Hypervisor] Data Storage or [Hypervisor] Instruction Storage interrupt will be caused.

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

Programming Note

The atomic setting of the Reference and Change bits enables an optimized sampling of them, for example when determining what pages to reclaim for other uses. To accurately sample the bits under HPT translation, it is necessary to first invalidate the PTE and the corresponding TLB entries. The optimized sequence eliminates the requirement for the relatively expensive invalidation of the TLB entries before sampling the bits. Instead, software may simply load the PTE using a Load And Reserve instruction, and then set the PTE invalid using a Store Conditional instruction. The TLB invalidation may be deferred indefinitely and grouped into cluster bombs for improved performance. The Reference and Change bits sampled in this manner are accurate (if the store conditional succeeds) because with the PTE marked invalid, it will be impossible to access a page for which the appropriate bit is not already set.

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.

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.
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 following kinds of interrupts:

- system-caused interrupts (see Section 6.4 on page 1057)
- 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.

---

**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.

When the hardware updates the Reference and Change bits in a Page Table Entry, the accesses are performed as described in Section 5.7.3.4, “Storage Control Attributes for Implicit Storage Accesses” on page 986. 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 5.10, “Translation Table Update Synchronization Requirements” on page 1043.

---

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 MSR_{DR}=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 5.10 for the rules software must follow when updating Reference and Change bits.

Figure 39 on page 1010 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".

### 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.

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.

### Status of Access

<table>
<thead>
<tr>
<th></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 MSR_{DR}=1</td>
<td>Acc^1</td>
<td>No</td>
</tr>
<tr>
<td>Storage protection violation</td>
<td>Acc</td>
<td>No</td>
</tr>
<tr>
<td>Out-of-order Store-type inst’n, including transactional Store-type inst’n, excluding dcbtst</td>
<td>Acc</td>
<td>Acc^1</td>
</tr>
<tr>
<td>Would be required by the sequential execution model in the absence of system-caused or imprecise interrupts^3, or transaction failure</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 transactional Load-type inst’n or dcbtst)</td>
<td>Acc</td>
<td>No</td>
</tr>
<tr>
<td>In-order Load-type or Store-type insn, access not performed^4</td>
<td>Acc</td>
<td>Acc^2</td>
</tr>
<tr>
<td>Store-type insn</td>
<td>Acc</td>
<td>No</td>
</tr>
<tr>
<td>Load-type insn</td>
<td>Acc</td>
<td>No</td>
</tr>
<tr>
<td>Other in-order access</td>
<td>Acc</td>
<td>No</td>
</tr>
<tr>
<td>Other ordinary Store, dcbz</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>icbi, icbt, dcbt, dcbtst, dcbst, dcbf disbelief</td>
<td>Acc</td>
<td>No</td>
</tr>
<tr>
<td>I-fetch or ordinary Load</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

^1 Acc means that it is acceptable to set the bit.
^2 It is preferable not to set the bit.
^3 If C is set, R is also set unless it is already set.
^4 For Floating-Point Enabled Exception type Program interrupts, "imprecise" refers to the exception mode controlled by MSR_{FE0,FE1}.
^4 This case does not apply to the Touch instructions, because they do not cause a storage access.

### Figure 39. Setting the Reference and Change bits
5.7.13 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 contents of one or more of the following:
- MSR bits HV, 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 storage protection mechanism consists of the Virtual Page Class Key Protection mechanism described in Section 5.7.13.1, the Basic Storage Protection mechanism described in Section 5.7.13.2 and Section 5.7.13.3, and the Radix Tree Translation Storage Protection mechanism described in Section 5.7.13.4.

When address translation is enabled for an access, the access is permitted in Paravirtualized HPT mode if and only if the access is permitted by both the Virtual Page Class Key Protection mechanism and the Basic Storage Protection mechanism. When address translation is enabled for a guest access, the access is permitted in Radix on Radix mode if and only if the access is permitted by the Radix Tree Translation Storage Protection mechanism for both the process-scoped and partition-scoped PTEs. When address translation is disabled for a guest access or is enabled for an access with MSRHV=1, the access is permitted in Radix on Radix mode if and only if the access is permitted by the Radix Tree Translation Storage Protection mechanism for the partition-scoped PTE. When address translation is disabled for an access with MSRHV=1, the access is permitted if and only if the access is permitted by the Basic Storage Protection mechanism. If an instruction fetch is not permitted, an Instruction Storage exception or a Hypervisor Instruction Storage exception is generated. If 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, virtual, or guest real addresses that cannot be mapped to real addresses. A protection boundary is a boundary between protection domains.

5.7.13.1 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 40, and to modify instruction access permissions for each class by modifying the Instruction Authority Mask Register (IAMR) shown in Figure 41.

**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.

**Authority Mask Register**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:1</td>
<td>Key0</td>
<td>Access mask for class number 0</td>
</tr>
<tr>
<td>2:3</td>
<td>Key1</td>
<td>Access mask for class number 1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>2n:2n+1</td>
<td>Keyn</td>
<td>Access mask for class number n</td>
</tr>
<tr>
<td>62:63</td>
<td>Key31</td>
<td>Access mask for class number 31</td>
</tr>
</tbody>
</table>

**Figure 40. Authority Mask Register (AMR)**

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 AMR\(2n\) and AMR\(2n+1\) refer to the first and second bits of the access mask corresponding to class number \(n\).

- A store is permitted if AMR\(2n\)=0b0; otherwise the store is not permitted.
- A load is permitted if AMR\(2n+1\)=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.
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.

### 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

<table>
<thead>
<tr>
<th>Bits</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:1</td>
<td>Key0</td>
<td>Access mask for class number 0</td>
</tr>
<tr>
<td>2:3</td>
<td>Key1</td>
<td>Access mask for class number 1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>2n:2n+1</td>
<td>Keyn</td>
<td>Access mask for class number n</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>62:63</td>
<td>Key31</td>
<td>Access mask for class number 31</td>
</tr>
</tbody>
</table>

Figure 41. Instruction Authority Mask Register (IAMR)

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 42 and Figure 43 respectively, can be used to restrict modifications (mtspr) 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.

![Figure 42. Authority Mask Override Register (AMOR)](image)

![Figure 43. User Authority Mask Override Register (UAMOR)](image)

The bits of the AMOR and UAMOR are in 1-1 correspondence with the bits of the AMR (i.e., [IAMORij corresponds to AMRIj). 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., AMOR2j+1 corresponds to IAMR2j+1). The AMOR affects modifications of the IAMR in privileged but non-hypervisor state; the IAMR cannot be accessed in problem state.

- When mtspr specifying the AMR (using either SPR 13 or SPR 29) or the IAMR is executed in privileged but 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 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 mtspr specifying the UAMOR is executed in privileged but 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 mtspr instruction in Section 4.4.4.

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 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.

---

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.)
Programming Note

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 PTE154 (which is either the AC bit or KEY4) to the same value that the AC bit would be set to, and set PTE12:3 (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 PTE152:53 (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.)
5.7.13.2 Basic Storage Protection, Address Translation Enabled

When address translation is enabled, the Basic Storage Protection mechanism is controlled by the following.

An example of the use of the AMOR (and UAMOR) is to support adjuncts (see Section 5.7.4, "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.)
MSR_{PR}, which distinguishes between supervisor (privileged) state and problem state
K_s and K_p, 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 5.7.8.1 and 5.7.9.2)
  - PTEG, 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 PTEG=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:
   \[ \text{Key} = (K_p \& \text{MSR}_{PR}) | (K_s \& \neg\text{MSR}_{PR}) \]

Using the computed Key, Figure 44 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 boundlessly undefined.

1. If MSR_{HV}=0, access authority is determined as described in Section 5.7.3.3.
2. If MSR_{HV}=1, the access is permitted.

5.7.13.4 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_{1:3} provide control over read, read/write, and execute access if the process has the appropriate privilege. EAA_{0}, 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_{0} permits the access. See Figure 35 and Figure 45 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 problem state for process-scoped protection of execute is determined by key 0 of the IAMR. When bit 1 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.
5.8 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.

5.8.1 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.

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.

- MSR bit IR or DR is 0 for instruction fetches or data accesses respectively, MSR_HV=1, and the storage is outside the range(s) specified by the Hypervisor Real Mode Storage Control facility (see Section 5.7.3.2.1).

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 Inexact 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 8.4.)

5.8.1.1 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}R=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_{R}=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).

5.8.2 Storage Control Bits

When the thread is not in hypervisor 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 46 and Figure 47. 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 5.7.10.3), except where otherwise stated or obvious from context.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Storage Control Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>W1,3</td>
<td>0 - not Write Through Required</td>
</tr>
<tr>
<td></td>
<td>1 - Write Through Required</td>
</tr>
<tr>
<td>I3</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.
3 The combination WIMG = 0b1110 has behavior unrelated to the meanings of the individual bits. See see Section 5.8.2.1, “Storage Control Bit Restrictions” for additional information.

**Figure 46. Storage control bits, HPT PTE**

<table>
<thead>
<tr>
<th>Att value</th>
<th>Storage Type</th>
<th>(WIMG=0010)</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>normal memory</td>
<td>(WIMG=0010)</td>
</tr>
<tr>
<td>01(^1)</td>
<td>SAO</td>
<td>(WIMG=1110)</td>
</tr>
<tr>
<td>10</td>
<td>non-idempotent I/O</td>
<td>(WIMG=0111)</td>
</tr>
<tr>
<td>11</td>
<td>tolerant I/O</td>
<td>(WIMG=0110)</td>
</tr>
</tbody>
</table>

W=0 always for Radix Tree translation
M=1 always for Radix Tree translation

1 Behaves like WIMG=0010 but with strong access order.

**Figure 47. Storage control bits, Radix PTE**

When the thread is not in hypervisor 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 5.7.13.

When the thread is in hypervisor real addressing mode, the storage control attributes are implicit; see Section 5.7.3.2.
In Sections 5.8.2.1 and 5.8.2.2, “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 real addressing mode.

5.8.2.1 Storage Control Bit Restrictions

All combinations of W, I, M, and G values are permitted except those for which both W and I are 1 and M||G ≠ 0b10.

The combination WIMG = 0b1110 is used to identify the Strong Access Ordering (SAO) storage attribute (see Section 1.7.1, “Storage Access Ordering", in Book II). Because this attribute is not intended for general purpose programming, it is provided only for a single combination of the attributes normally identified using the WIMG bits. That combination would normally be indicated by WIMG = 0b0010.

References to Caching Inhibited storage (or storage with I=1) elsewhere in the Power ISA have no application to SAO storage or its WIMG encoding, despite the fact that the encoding uses I=1. Conversely, references to storage that is not Caching Inhibited (or storage with I=0) apply to SAO storage or its WIMG encoding. References to Write Through Required storage (or storage with W=1) elsewhere in the Power ISA have no application to SAO storage or its WIMG encoding, despite the fact that the encoding uses W=1. Conversely, references to storage that is not Write Through Required (or storage with W=0) apply to SAO storage or its WIMG encoding.

If a given real page is accessed concurrently as SAO storage and as non-SAO storage, the result may be characteristic of the weakly consistent model.

At any given time, the value of the W bit must be the same for all accesses to a given real page.

5.8.2.2 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 \texttt{dcbst} or \texttt{dcbf[l]}.

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 \texttt{dcbf[l]} and \texttt{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 4.4.1 on page 965.)

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. The operating system should provide a means by which application programs can request SAO storage, in order to avoid confusion with the preceding guideline (since SAO is encoded using WI=0b11).

At any given time, the value of the I bit must be the same for all accesses to a given real page.

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.

At any given time, the value of the I bit must be the same for all accesses to a given real page.

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 \texttt{dcbst} or \texttt{dcbf[l]}.

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 \texttt{dcbf[l]} and \texttt{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 4.4.1 on page 965.)

The storage control bit alterations described above are examples of cases in which the directives for application of statements about the W and I bits to SAO given in the third paragraph of the preceding subsection must be applied. A transition from the typical WIMG=0b0010 for ordinary storage to WIMG=0b1110 for SAO storage does not require the flush described above because both WIMG combinations indicate storage that is not Caching Inhibited.

It is recommended that \texttt{dcbf} be used, rather than \texttt{dcbfl}, when changing the value of the I or W bit from 0 to 1. (\texttt{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, \texttt{dcbf} need be executed on only one thread in the system.)

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.

For example, when changing the M bit in some directory-based systems, software may be required to execute \texttt{dcbf[l]} on each thread to flush all storage locations accessed with the old M value before permitting the locations to be accessed with the new M value.
Additional requirements for changing the storage control bits in the Page Table are given in Section 5.10.
5.9 Storage Control Instructions

5.9.1 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 5.6) 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.

5.9.2 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 and hypervisor 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` or `msgsnd` instruction that follows the `Synchronize` instruction. The thread that is the target of the `msgsndp` or `msgsnd` instruction is here called the "target thread".

- For `msgsndp`, and L = 0, 1, or 2 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`, and L = 0 or 2 for the `Synchronize` instruction (`sync` or `ptesync`), the stores are performed with respect to any given other thread before any data accesses caused by instructions that are executed on the given thread after a `msgsync` instruction is executed on that thread after the corresponding Directed Hypervisor Doorbell interrupt has occurred.

**Programming Note**

The `msgsnp` instruction, which is needed when `msgsnd` 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` can target any thread in the system. (If the target thread for `msgsnd` is on the same multi-threaded processor as the thread executing the `msgsnd`, in principle the `msgsync` can be omitted. This optimization is practical only when the `msgsnd` topology is appropriately constrained, however, because the Directed Hypervisor Doorbell interrupt provides no indication of which thread executed the `msgsnd` 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 `ptesync`) provides the appearance of "causality" across a sequence of `msgsnd` 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.

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 **ptesync** (equivalent to **sync** with L=2).

The **ptesync** instruction has all of the properties of **sync** with L=0 and also the following additional properties:

- The memory barrier created by the **ptesync** instruction provides an ordering function for the storage accesses associated with all instructions that are executed by the thread executing the **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 **ptesync** instruction, before the **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 **ptesync** instruction causes all Reference and Change bit updates associated with address translations that were performed, by the thread executing the **ptesync** instruction, before the **ptesync** instruction is executed, to be performed with respect to that thread before the **ptesync** instruction’s memory barrier is created.

- The memory barrier created by the **ptesync** instruction provides an ordering function for all stores to the Partition Table, Process Tables, Segment Tables, Page Directories, and Page Tables caused by **Store** instructions preceding the **ptesync** instruction with respect to invalidations, of cached copies of information derived from these tables, caused by **slibie**, **slibiag**, and **tlbie** instructions following the **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 **slibie**, **slibiag**, or **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).

**Programming Note**

The next bullet is sufficient to order the stores with respect to the invalidations on the thread executing the **ptesync** instruction. That bullet is also sufficient to provide the ordering with respect to invalidations caused by **slibie**, **slibia**, and **tlbie** instructions, which affect only the thread executing them.

- The **ptesync** instruction provides an ordering function for all stores to the Partition Table, Process Tables, Segment Tables, Page Directories, and Page Tables caused by **Store** instructions preceding the **ptesync** instruction with respect to searches of these tables that are performed, by the thread executing the **ptesync** instruction, after the **ptesync** instruction completes. Executing a **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 **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).

- In conjunction with the **tlbie** and **tlbsync** instructions, the **ptesync** instruction provides an ordering function for TLB invalidations and related storage accesses on other threads as described in the **tlbsync** instruction description on page 1042.

Similarly, in conjunction with the **slibie** or **slibia** and **slbsync** instructions, the **ptesync** instruction provides an ordering function for SLB invalidations and related storage accesses on other threads as described in the **slbsync** instruction description on page 1032.

**Programming Note**

For instructions following a **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 **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 **sync** with L=0 should be used (or **sync** with L=1, or **eieio**, if appropriate).

Section 5.10, “Translation Table Update Synchronization Requirements” on page 1043 gives examples of uses of **ptesync**.

**5.9.3 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_UPT), 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 Loo-
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 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.

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 5.10.1.2.

The effects of the slbie, slbieg, slbia, slbiag, and TLB Management instructions on address translations, as specified in Sections 5.9.3.2 for the SLB and 5.9.3.3 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 and HRMOR, 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 5.9.3.2 - 5.9.3.3 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 11.

---

### 5.9.3.1 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 slbmt to install such translations in the SLB. All SLB entries created using slbmt are considered to be “software created.” Software created entries will only translate accesses from the hardware thread by which they are installed. When LPCRupt=1, they are also considered to be “bolted.” Each thread has the ability to bolt four entries.

---

### 5.9.3.2 SLB Management Instructions

Software establishes translations in the SLB using slbmt. 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 slbmt. 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 is the only way to invalidate outboard translation caches. When taking a PID out of service with the intent of reusing it, software should use slbiag to remove stale translations from SLBs and ERATs in the "nest." (Nest
refers to the platform external to the processor cores. Here the reference is to translations cached for use by accelerators.) \textit{slbsync} will establish order between \textit{slbieg} and \textit{slbiag} instructions and a subsequent \textit{ptesync}. \textit{ptesync} must also be used to synchronize the Segment Table update prior to performing the lookaside management. When performing a context switch, software must use an \textit{slbia} instruction to remove lookaside information associated with the old context. \textit{slbmfee} and \textit{slbmfev} may be used by the hypervisor to save software-created entries. \textit{slbmte} is used to restore software-created entries. \textit{slbfee} has no function when LPCR\textsubscript{UPRT}=1 for the partition that is running.

**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 11.

---

**SLB Invalidate Entry**

| X-form | 
|-----|-----|-----|-----|-----|-----|
| 0   | 31 | 30 | 29 | 28 | 27 |
| 6   | 5  | 4  | 3  | 2  | 1  |
| 16  | 15 | 14 | 13 | 12 | 11 |
| 21  | 20 | 19 | 18 | 17 | 16 |
| 31  | 30 | 29 | 28 | 27 | 26 |

- \(\text{slbie} \text{ RB}\)

\(\text{e}_a:35 \leftarrow (\text{RB}_0:35)\)

if, for SLB entry that translates

- most recently translated \(e_a\)

\(\text{entry}\_\text{seg}\_\text{size} = \text{size specified in } (\text{RB})_{37:38}\)

then for SLB entry (if any) that translates \(e_a\)

- \(\text{SLBE}_v \leftarrow 0\)

- \(\text{all other fields of } \text{SLBE} \leftarrow \text{undefined}\)

else

- \(s \leftarrow \text{log}_2(\text{entry}\_\text{seg}\_\text{size})\)

- \(\text{esid} \leftarrow (\text{RB})_{0:63-s}\)

- \(u \leftarrow \text{undefined 1-bit value}\)

- if \(u\) then

- if an SLB entry translates \(\text{esid}\)

- \(\text{SLBE}_v \leftarrow 0\)

- \(\text{all other fields of } \text{SLBE} \leftarrow \text{undefined}\)

Let the Effective Address (EA) be any EA for which \(\text{EA}_0:35 = (\text{RB}_0:35)\). Let the segment size be equal to the segment size specified in \(\text{RB})_{37:38}\); the allowed values of \(\text{RB})_{37:38}\), and the correspondence between the values and the segment size, are the same as for the B field in the SLBE (see Figure 27 on page 994).

The segment size must be the same as the 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.

- \((\text{RB})_{37}\)
- \((RB)_{39}\)
- \((RB)_{40:63}\)
- 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**

\textit{slbie} does not affect SLBs on other threads.

---

**Programming Note**

The \(B\) value in register \(RB\) may be needed for invalidating ERAT entries corresponding to the translation being invalidated.

---

**Programming Note**

When switching to execute an adjunct, a hypervisor will disable translation and use \textit{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 Segment Table walks will be performed.

---

**SLB Invalidate Entry Global**

\textit{x-form}

\textit{slbieg} RS, RB

\[
\begin{array}{cccc}
31 & RS & /// & RB & 466 \mid \\
0 & 6 & 11 & 16 & 21 & 31
\end{array}
\]

\begin{itemize}
\item target\_PID = RS_{0:31}
\item if MSR\_hyp\_en = 1 then target\_LPID = RS_{32:63}
\item else target\_LPID = LPIDR
\item \(ea_{0:35} \leftarrow (RB)_{0:35}\)
\item for each thread with LPIDR = target\_LPID and PIDR = target\_PID
\item if, for each SLB entry that translates or most recently translated \(ea\)
\item entry\_seg\_size = size specified in \((RB)_{37:38}\)
\item then for SLB entry (if any)
\item that translates \(ea\) and is not software-created
\item SLBE\_V \leftarrow 0
\item all other fields of SLBE \leftarrow undefined
\item else
\item \(s \leftarrow \text{log}\_\text{base}\_2(\text{entry}\_seg\_size)\)
\item \(esid \leftarrow (RB)_{0:63} - s\)
\item \(u \leftarrow \text{undefined}\) 1-bit value
\item if \(u\) then
\item if an SLB entry translates esid and the entry is not software-created
\item SLBE\_V \leftarrow 0
\item all other fields of SLBE \leftarrow undefined
\end{itemize}

---

The operation performed by this instruction is based on the contents of registers \(RS\) and \(RB\). The contents of these registers are shown below.

**RS**

\[
\begin{array}{cc}
\text{PID} & \text{LPID} \\
0 & 32 & 63
\end{array}
\]

**RB**

\[
\begin{array}{cccc}
\text{ESID} & C & B & 0s \\
0 & 36 & 37 & 39 & 63
\end{array}
\]

- \(RS_{0:31}\) PID
- \(RS_{32:63}\) LPID
- \(RB_{0:35}\) ESID
- \(RB_{36}\) must be 0b0
- \(RB_{37:38}\) B
- \(RB_{39:63}\) must be 0b0 || 0x0000000

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. Let the Effective Address (EA) be any EA for which \(EA_{0:35} = (RB)_{0:35}\); 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 27 on page 994).

Only SLBs for threads running on behalf of target_LPID and target_PID are searched. Software-created entries are ignored. The segment size must be the same as the 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)36
- (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 **sync** 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_G-TSE=0, making it hypervisor privileged.

**Special Registers Altered:**

None

---

**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 be ignored by the instruction.

---

**SLB Invalidate All**

**X-form**

```
slbia IH
```

| 31 | 8 | 11 | 16 | 21 | 498 | 31 |

```
switch (IH)
case (0b000, 0b001, 0b010, 0b110):
  for each SLB entry except SLB entry 0
  SLBE = 0
  all other fields of SLBE ← undefined

case (0b011):
  for each SLB entry such that SLBClass = 1
  SLBEV = 0
  all other fields of SLBE ← undefined

case (0b100):
  for each SLB entry
  SLBE = 0
  all other fields of SLBE ← undefined

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,” and “when address translation was enabled” is shorthand for “when MSRIR was equal to 1 or MSRDR was equal to 1, as appropriate for the type of access,” and correspondingly for “when address translation was disabled.” The descriptions specify which entries must be invalidated; additional entries may be invalidated except where the description states that certain SLB entries are not invalidated.

0b000  All SLB entries except entry 0 are invalidated; SLB entry 0 is not invalidated.
All implementation-specific lookaside information is invalidated.

---

**Programming Note**

**slbia** does affect SLBs on other threads.

---

The **slbieg** instruction to invalidate software-created segment descriptors is a programming error. The architecture requires that bolted entries be ignored by the instruction.

---

**Programming Note**

The B value in register RB may be needed for invalidating ERAT entries corresponding to the translation being invalidated.

---

The **slbieg** instruction does not affect SLBs on other threads.
0b001 All SLB entries except entry 0 are invalidated; SLB entry 0 is not invalidated. All implementation-specific lookaside information that was created when address translation was enabled and satisfies either of the following conditions is invalidated:
- 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 ≠ 0.

0b010 All SLB entries except entry 0 are invalidated; SLB entry 0 is not invalidated. All implementation-specific lookaside information that was created when address translation was enabled is invalidated.

0b011 All SLB entries having a Class value of 1 are invalidated; SLB entry 0 is not invalidated. All implementation-specific lookaside information that was created when address translation was enabled and satisfies either of the following conditions is invalidated:
- 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 ≠ 0.

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 ≠ 0 or effPID ≠ 0.
- 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**

When performing a context switch between processes, an HPT operating system will use `mtLPIDR` followed by `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 `mtLPIDR`, and then perform the `slbia` on all the affected threads. (If the affected threads were not in real mode, avoiding implicit branches due to the `mtLPIDR` would be very difficult.)

**Programming Note**

`slbia` does not affect SLBs on other threads.

**Programming Note**

If `slbia` is executed when instruction address translation is enabled, software can ensure that attempting to fetch the instruction following the `slbia` does not cause an Instruction Segment interrupt by placing the `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 `slbia` and executing the subsequent instruction. It also assumes that IH values other than 0b011 and 0b100 are used.)

---

All other IH values are reserved. If the IH field contains a reserved value, the hint provided by the IH field is undefined.

---

In the preceding description, “SLB-derived translation” excludes any SLS translation, since SLS translation does not use segmentation.

When IH=0b000, 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 5.7.3.2.1, “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
SLB Invalidate All Global X-form

slbiag RS

\[
\begin{array}{|c|c|c|c|c|}
\hline
& RS & \text{///} & \text{///} & 850 \\
\hline
0 & 31 & 8 & 11 & 16 & 21 & 31 \\
\hline
\end{array}
\]

target_PID = RS_{0:31}
if MSR_{hyp}=1 then target_LPID = RS_{32:63}
else target_LPID = LPIDR
for each nest SLB
   for each SLBE with LPID=target_LPID and PID=target_PID
      SLBE_y \leftarrow 0
      all other fields of SLBE \leftarrow undefined

The operation performed by this instruction is based on the contents of register RS. The contents of this register is shown below.

RS

\[
\begin{array}{|c|c|}
\hline
\text{PID} & \text{LPID} \\
\hline
0 & 32 & 63 \\
\hline
\end{array}
\]

RS_{0:31} PID
RS_{32:63} LPID

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. Each SLBE for process PID in 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.

Implementation specific lookaside information associated with the invalidated SLB entries is invalidated. Additional implementation specific lookaside information may be invalidated.

The operation performed by this instruction is ordered by the \textit{eieio} (or \textit{sync} or \textit{ptesync}) instruction with respect to a subsequent \textit{slbsync} instruction executed by the thread executing the \textit{slbiag} instruction. The operations caused by \textit{slbiag} and \textit{slbsync} are ordered by \textit{eieio} as a fifth set of operations, which is independent of the other four sets that \textit{eieio} orders.

This instruction is privileged except when LPCR_{G-}
TSE=0, making it hypervisor privileged.

**Special Registers Altered:**

None
When LPCR<sub>UPRT</sub>=0, this instruction is the sole means for specifying Segment translations to the hardware. When LPCR<sub>UPRT</sub>=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 48.

RS

<table>
<thead>
<tr>
<th>B</th>
<th>VSID</th>
<th>Ks</th>
<th>Kp</th>
<th>N</th>
<th>L</th>
<th>C</th>
<th>must be 0b0</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>
<td></td>
<td></td>
<td></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>

**Figure 48. 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<sub>UPRT</sub>=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)60:63
- (RB)37:51

If this instruction is executed in 32-bit mode, (RB)10:31 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_{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 5.7.8.2, "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></td>
</tr>
<tr>
<td>[30]</td>
<td></td>
</tr>
<tr>
<td>[29]</td>
<td></td>
</tr>
<tr>
<td>[28]</td>
<td>[L]</td>
</tr>
<tr>
<td>[27]</td>
<td>[RB]</td>
</tr>
<tr>
<td>[26]</td>
<td>851</td>
</tr>
<tr>
<td>[25]</td>
<td></td>
</tr>
</tbody>
</table>

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 B, VSID, \( K_s \), \( K_p \), 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 49.

#### RT

<table>
<thead>
<tr>
<th>B</th>
<th>VSID</th>
<th>( K_s )</th>
<th>( K_p )</th>
<th>N</th>
<th>L</th>
<th>C</th>
<th>LP</th>
<th>Os</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>52</td>
<td>57</td>
<td>60</td>
<td>63</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### RB

<table>
<thead>
<tr>
<th>Os</th>
<th>index</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>52</td>
</tr>
</tbody>
</table>

| RT_{0:1} | B |
| RT_{2:51} | VSID |
| RT_{52} | \( K_s \) |
| RT_{53} | \( K_p \) |
| RT_{54} | N |
| RT_{55} | L |
| RT_{56} | C |
| RT_{57} | set to 0b0 |
| RT_{58:59} | LP |
| RT_{60:63} | set to 0b0000 |

| RB_{0:51} | must be 0x0_0000_0000_0000 |
| RB_{52:63} | index, which selects the SLB entry |

**Figure 49. GPR contents for slbmfev**

On implementations that support a virtual address size of only n bits, n<78, RT_{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)_{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_{0:51}.

This instruction is privileged.

---

The use of the L field is implementation specific.

**Special Registers Altered**:

None
### SLB Move From Entry ESID X-form

**slbmf**ee \( RT, RB \)

<table>
<thead>
<tr>
<th>31</th>
<th>RT</th>
<th>///</th>
<th>L</th>
<th>RB</th>
<th>915</th>
<th>/</th>
</tr>
</thead>
</table>

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 50.

#### RT

<table>
<thead>
<tr>
<th>31</th>
<th>ESID</th>
<th>V</th>
<th>BO</th>
<th>0</th>
</tr>
</thead>
</table>

#### RB

<table>
<thead>
<tr>
<th>0</th>
<th>index</th>
</tr>
</thead>
</table>

**Figure 50. GPR contents for slbmf**ee

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 \(RB_{0:51}\).

This instruction is privileged.

#### The use of the L field is implementation specific.

#### Special Registers Altered:

None

### SLB Find Entry ESID X-form

**slb**fe. \( RT, RB \)

<table>
<thead>
<tr>
<th>31</th>
<th>RT</th>
<th>///</th>
<th>L</th>
<th>RB</th>
<th>979</th>
<th>1</th>
</tr>
</thead>
</table>

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 \(SLBE[ESID_{0:63-s}]\) (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, \(K_s\), \(K_p\), \(N\), \(L\), \(C\), 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 51.

#### RT

<table>
<thead>
<tr>
<th>B</th>
<th>VSID</th>
<th>(K_sK_p)</th>
<th>NLC</th>
<th>0</th>
<th>LP</th>
<th>0s</th>
</tr>
</thead>
</table>

#### RB

<table>
<thead>
<tr>
<th>0</th>
<th>index</th>
</tr>
</thead>
</table>

**Figure 51. GPR contents for slb**fe.

If \(s > 28\), \(RT_{80-s:51}\) are set to zeros. On implementations that support a virtual address size of only \(n\) bits, \(n < 78\), \(RT_{2:79-n}\) 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, \(j=0b0\).
CR0_LT_GT_EQ_SO = 0b00 || j || XER_SO

The hardware ignores the contents of RB[36:38 40:63].

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 is privileged.

**Special Registers Altered:**

CR0

---

**SLB Synchronize**

| CR0_LT_GT_EQ_SO = 0b00 || j || XER_SO |
|---|

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 **sync** 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_G-TSE=0, making it hypervisor privileged.

See Section 5.10 for a description of other requirements associated with the use of this instruction.

**Special Registers Altered:**

None

---

**Programming Note**

**slbsync** should not be used to synchronize the completion of **slbie**.
5.9.3.3 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 series of consecutive 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_WR=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_HV=1, invalidate all entries, otherwise invalidate matching LPID.

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 series of consecutive 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_WR=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_HV=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. The IS=0 `tlbie` variants operate on a specified congruence class, requiring a software loop where `tlbie` operates on the entire TLB. For IS=0 invalidations of Radix Tree translations, the use of `tlbie[1]` 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.

---

**Programming Note**

Changes to the Page Table in the presence of active transactions may compromise transactional semantics if a page accessed by a translation is remapped within the lifetime of the transaction. Through the use of a `tlbie` instruction to the unmapped page, an operating system or hypervisor can ensure that any transaction that has touched the affected page is terminated.

Changes to local translation lookaside buffers, through the `tlbiel` instruction, have no effect on transactions. Consequently, if these instructions are used to invalidate TLB entries after the unmapping of a page, it is the responsibility of the OS or hypervisor to ensure that any transaction that may have touched the modified page is terminated, using a `tabort` or `treclaim` instruction.
TLB Invalidate Entry  

X-form

\[\text{tlbie RB,RS,RIC,PRS,R} \]

\[L \leftarrow (RB)_{52:53}\]

if MSRHV=1 then search_LPID=RS_{32:63}
else search_LPID=LPIDRLPID

\begin{align*}
\text{switch}(IS) & \\
\text{case (0b00)}: & \\
& \text{if RIC=0} & \\
& \text{if R=0 then} & \\
& \text{then} & \\
& \text{base pg size} = 4X & \\
& \text{actual pg size} = & \\
& \text{page size specified in (RB)_{56:58}} & \\
& i = 51 & \\
& \text{else} & \\
& \text{base pg size} = & \\
& \text{base page size specified in (RB)_{44:51}} & \\
& \text{actual pg size} = & \\
& \text{actual page size specified in (RB)_{44:51}} & \\
& b = \log_{2}(\text{base pg size}) & \\
& p = \log_{2}(\text{actual pg size}) & \\
& i = \max(\min(43,63-b),63-p) & \\
& \text{sg size = segment size specified in (RB)_{54:56}} & \\
& \text{for each thread} & \\
& \text{for each TLB entry} & \\
& \text{if (entry VA_{14:1+14} = (RB)_{0:i})} & \\
& \text{& (entry sg size = sg size)} & \\
& \text{& (entry base pg size = base pg size)} & \\
& \text{& (entry actual pg size = actual pg size)} & \\
& \text{& (entry LPID = search LPID)} & \\
& \text{& (entry process scoped = 0)} & \\
& \text{then} & \\
& \text{if ((L = 0) | (b \geq 20))} & \\
& \text{then TLB entry \leftarrow invalid} & \\
& \text{else} & \\
& \text{if (entry VA_{58:77-b} = (RB)_{56:75-b})} & \\
& \text{then TLB entry \leftarrow invalid} & \\
& \text{else} & \\
& \text{actual pg size} = & \\
& \text{page size specified in (RB)_{56:58}} & \\
& p = \log_{2}(\text{base pg size}) & \\
& i = 63-p & \\
& \text{for each thread} & \\
& \text{for each TLB entry} & \\
& \text{if (entry RA_{5:4} = (RB)_{0:1})} & \\
& \text{& (entry actual pg size = actual pg size)} & \\
& \text{& (entry LPID = search LPID)} & \\
& \text{& (entry process scoped = PRS)} & \\
& \text{& ((PRS = 0) | (entry PID = (RS)_{0:31}))} & \\
& \text{then} & \\
& \text{TLB entry \leftarrow invalid} & \\
& \text{else if RIC=3 then} & \\
& \text{if RB_{AD} indicates cluster bomb then} & \\
& n = \text{implementation-specific series size} & \\
& f(L | AP) & \\
& \text{base pg size} = & \\
& \text{implementation-specific base page size} & \\
& f(L | AP) & \\
& \text{trunc} = \log_{2}(n \times \text{base pg size}) - 12 & \\
& \text{loop RB \leftarrow RB} & \\
& \text{loop RB_{AD} \leftarrow implementation-specific} & \\
& \text{encoding for base pg size} & \\
& \text{loop RB_{VPN} \leftarrow loop RB_{VPN}(0:51-\text{trunc}) || trunc0} & \\
& \text{do i=0 to n-1} & \\
& \text{tlbie loop RB,RS,0,0,0} & \\
& \text{loop RB_{VPN} + (base pg size/4096)} & \\
& \text{case (0b01):} & \\
& \text{if RIC=0 | RIC=2 then} & \\
& \text{for each TLB entry for each thread} & \\
& \text{if (entry LPID=search LPID)} & \\
& \text{& (entry PID=RS_{0:31})} & \\
& \text{& (entry PRS=1)} & \\
& \text{then TLB entry \leftarrow invalid} & \\
& \text{if RIC=1 | RIC=2 then} & \\
& \text{for each thread} & \\
& \text{invalidate process-scoped radix page walk} & \\
& \text{caching associated with process RS_{0:31} in} & \\
& \text{partition search LPID} & \\
& \text{if (RIC=2) \& (PRS=1) then} & \\
& \text{for each thread} & \\
& \text{invalidate Process Table caching associated} & \\
& \text{with process RS_{0:31} in partition search LPID} & \\
& \text{case (0b10):} & \\
& \text{if RIC=0 | RIC=2 then} & \\
& \text{if (PRS=0) \& (MSRHV=1) \& (R=0)) then} & \\
& \text{for each partition-scoped TLB entry for each thread} & \\
& \text{if entry LPID=search LPID} & \\
& \text{then TLB entry \leftarrow invalid} & \\
& \text{if PRS=1 then} & \\
& \text{for each process-scoped TLB entry for each thread} & \\
& \text{if entry LPID=LPID} & \\
& \text{then TLB entry \leftarrow invalid} & \\
& \text{if RIC=1 | RIC=2 then} & \\
& \text{for each thread} & \\
& \text{for each thread invalidate partition-scoped} & \\
& \text{page walk caching associated with} & \\
& \text{partition search LPID} & \\
& \text{if (PRS=0) \& (MSRHV=1) then} & \\
& \text{for each thread invalidate Partition Table} & \\
& \text{caching associated with partition} & \\
& \text{search LPID} & \\
& \text{if PRS=1 then} & \\
& \text{for each thread invalidate Process Table} & \\
& \text{caching associated with partition} & \\
& \text{search LPID} & \\
& \text{case (0b11):} & \\
& \text{if RIC=0 | RIC=2 then} & \\
& \text{if MSRHV then} & \\
& \text{for all threads} & \\
& \text{all partition-scoped TLB entries \leftarrow invalid} & \\
& \end{align*}
else
dlall process-scoped TLB entries ← invalid
if (MSRHV=0) & (PRS=1) then
for each process-scoped TLB entry for each thread
if TLBELPID=search_LPID
then TLB entry ← invalid

if (MSRHV=0) & (PRS=0) & (R=0) then
for each partition-scoped TLB entry for each thread
if TLBELPID=search_LPID
then TLB entry ← invalid
if RIC=1 | RIC=2 then
if MSR[VH] then
if PRS=0 then
for all threads
invalidate all partition-scoped page walk caching
else
for all threads
invalidate all process-scoped page walk caching
if (MSR[VH]=0) & (PRS=1) then
for each thread invalidate process-scoped page walk caching associated with partition search_LPID
if RIC=2 then
if MSR[VH] then
if PRS=0 then
for each thread
invalidate all Partition Table caching
else
for each thread
invalidate all Process Table caching
if (MSR[VH]=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>

Programming Note

Note that although there is no PID compare for partition-scoped translation, software must still place the PID in RS. It may be used, for example, in the TLB hash.

RB for R=1 and IS=0b000:

<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=0b000, 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>

RB for R=0, IS=0b000, 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>0</td>
<td>44</td>
<td>52</td>
<td>54</td>
<td>56</td>
<td>63</td>
</tr>
</tbody>
</table>

RB for IS=0b01, 0b10, or 0b11:

<table>
<thead>
<tr>
<th>0s</th>
<th>IS</th>
<th>0s</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>52</td>
<td>54</td>
</tr>
</tbody>
</table>

If this instruction is executed in hypervisor state, RS[32:63] contains the partition ID (LPID) of the partition for which one or more translations are being invalidated. Otherwise, the value in LPIDR is used. The supported (RS)[32:63] values are the same as the LPID values supported in LPIDR. RS[0:31] contains a PID value. The supported values of RS[0:31] are the same as the PID values supported in PIDR.

The following forms are invalid.
- PRS=1, R=0, and RIC≠2 (The only process-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 (Cluster bombs are only supported for HPT translation.)

The following forms are treated as if the instruction form were invalid.
- RIC=1 and IS=0 (The architecture does not support shutdown of individual translations in the Page Walk Cache.)
- RIC=2 and IS=0 (RIC is for comprehensive invalidation that is not supported at the level of an individual page.)
- RIC=3 and IS=0 (Cluster bombs are only supported 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 incremental 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 EA[0:1]=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 SLBELLP 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 RB₄₄:₅₁ (LP - Large Page size selector field) and the base page size and actual page size is the same as the relationship between PTE₉₅ and the base page size and actual page size, except for the "+" bits (see Section 5.7.9.1 on page 998 and Figure 32 on page 999). Thus, b is equal to 12 and p is equal to log₂ (actual page size specified by RB₄₄:₅₁) and p is equal to log₂ (actual page size specified by RB₄₄:₅₁). Specifically, RB₄₄+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 maximum of 0 and (20-p).

- Variable i is the larger of (63-p) and the value that is the smaller of 43 and (63-b). (RB)ₜₖ 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)₅₆:₇₅ 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 RB₅₄:₅₅ (B field). The contents of RB₅₄:₅₅ 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.

RB₅₂:₅₃ and RB₅₉:₆₂ (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 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, VA₅₈:₇₇-b is equal to (RB)₅₆:₇₅-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.
  - TLBELPID = search_LPID.

Additional TLB entries may also be made invalid if those TLB entries contain an LPID that matches search_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 on all threads that have all of the following properties are made invalid.

- 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.
- 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 an implementation-specific encoding of AP indicates the number and (base) size of a series of sequential virtual pages for which the translations will be invalidated. The pages occupy an aligned region of virtual storage. The address in RB is masked to get the base address of the region. *tlbie* are then performed for the virtual pages within the region.

**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 TLBE_{LPID} 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_{PDP}=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 0b01, 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, RB_{40: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 RB_{63}=0, or RB44:51 when RB_{63}=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 *sync* or *ptesync*) 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_{GTEE}=0 or when PRS=0 and HR=1, making it hypervisor privileged.

See Section 5.10, “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><em>tlbie</em> RB,RS</td>
<td><em>tlbie</em> RB,RS,0,0,0</td>
</tr>
</tbody>
</table>

**Special Registers Altered:**

None
TLB Invalidate Entry Local X-form

**TLB Invalidate Entry Local X-form**

<table>
<thead>
<tr>
<th>Tlbie</th>
<th>RB, RS, RIC, PRS, R</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>31</td>
</tr>
<tr>
<td></td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>14</td>
</tr>
<tr>
<td></td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>21</td>
</tr>
<tr>
<td></td>
<td>274</td>
</tr>
<tr>
<td></td>
<td>31</td>
</tr>
</tbody>
</table>

**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 RB_{57:58}. 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 RB_{57:58}.

**Programming Note**

For `tlbie` instructions the AVA and AVAL fields in RB contain different VA bits from those in PTEAVA.

**Programming Note**

An operating system that uses HPT translation should only use `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 `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.

```
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)
seg_size ← segment size specified in (RB)_{54:55}
for each TLB entry
    if (entry_VA_{14:i+14} = (RB)_{0:i}) &
        (entry_sg_size = seg_size) &
        (entry_base_pg_size = base_pg_size) &
        (entry_actual_pg_size = actual_pg_size) &
        (TLBELPID = search_LPID) &
        (entry_process_scoped = 0)
    then
    if ((L = 0) | (b ≥ 20)) then
        TLB entry ← invalid
    else
        if (entry_VA_{58:77-b} = (RB)_{56:77-b}) then
            TLB entry ← invalid
    end
end
```

**Programming Note**

For `tlbie` 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 `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 `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.

```
for each TLB entry
    if (entry_VA_{14:i+14} = (RB)_{0:i}) &
        (entry_sg_size = seg_size) &
        (entry_base_pg_size = base_pg_size) &
        (entry_actual_pg_size = actual_pg_size) &
        (TLBELPID = search_LPID) &
        (entry_process_scoped = 0)
    then
    if ((L = 0) | (b ≥ 20)) then
        TLB entry ← invalid
    else
        if (entry_VA_{58:77-b} = (RB)_{56:77-b}) then
            TLB entry ← invalid
    end
end
```

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 RB_{57:58}. 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 RB_{57:58}.

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 RB_{57:58}. 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 RB_{57:58}.
if RIC=1 | RIC=2 then
  for each thread
    if (PRS=0)&(MSRHV=1) then
      invalidate partition-scoped page walk caching associated with partition search_LPID
    if PRS=1 then
      invalidate process-scoped page walk caching associated with partition search_LPID
  if RIC=2 then
    if (PRS=0)&(MSRHV=1) then
      invalidate Partition Table caching associated with partition search_LPID
    if PRS=1 then
      invalidate Process Table caching associated with partition search_LPID
  case (0b11):
    if RIC=0 | RIC=2 then
      i=implementation-dependent number, 40≤i≤51
    if MSRy0 then
      if PRS=0 then
        all partition-scoped TLB entries in set (RB)_{i:51} ← invalid
      else
        all process-scoped TLB entries in set (RB)_{i:51} ← invalid
    if MSRy0 & (PRS=1) then
      for each process-scoped TLB entry in set (RB)_{i:51}
      if entry_LPID=search_LPID
        then TLB entry ← invalid
        if (MSRHV=0)&(PRS=1) then
          for each process-scoped TLB entry in set (RB)_{i:51}
          if entry_LPID=search_LPID
            then TLB entry ← invalid

  if RIC=1 | RIC=2 then
    if MSRy0 then
      if PRS=0 then
        invalidate all partition-scoped page walk caching
      else
        invalidate all process-scoped page walk caching
    if MSRy0 & (PRS=1) then
      invalidate process-scoped page walk caching associated with partition search_LPID

  if RIC=2 then
    if MSRy0 then
      if PRS=0 then
        invalidate all Partition Table caching
      else
        invalidate all Process Table caching
    if MSRy0 & (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)_{52:31} and L is (RB)_{63}.

### Programming Note

Note that although there is no PID compare for partition-scoped translation, software must still place the PID in RS. It may be used, for example, in the TLB hash.

<table>
<thead>
<tr>
<th>PID</th>
<th>32</th>
<th></th>
<th>63</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>32</td>
<td></td>
<td>63</td>
</tr>
</tbody>
</table>

### RS:

#### 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>

#### RB for R=0, IS=0b00, 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>0</td>
<td>44</td>
<td>52</td>
<td>54</td>
<td>56</td>
<td>63</td>
</tr>
</tbody>
</table>

#### RB for IS=0b01, 0b10, or 0b11:

<table>
<thead>
<tr>
<th>0s</th>
<th>SET</th>
<th>IS</th>
<th>0s</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>40</td>
<td>52</td>
<td>54</td>
</tr>
</tbody>
</table>

LPIDR contains the partition ID (LPID) of the partition for which the translation is being invalidated. RS_{30:31} contains a PID value. The supported values of RS_{30:31} are the same as the PID values supported in PIDR.

The following forms are invalid.

- PRS=1, R=0, and RIC≠2 (The only process-scoped HPT caching is of the Process Table.)
- RIC=1 and R=0 (There is no Page Walk Cache for HPT translation.)
- RIC=3 (Cluster bombs are not supported for tlbei.)

The following forms are treated as though the instruction form was invalid.

- RIC=1 and IS=0 (The architecture does not support shootdown of individual translations in the Page Walk Cache.)
- RIC=2 and IS=0 (RIC is for comprehensive invalidation that is not supported at the level of an individual 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 incremental 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 SLBE = LP 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 PTELIp and the base page size and actual page size, except for the "r" bits (see Section 5.7.9.1 on page 998 and Figure 32 on page 999). 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 maximum of 0 and (20-p).

- 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+44 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)54:55 (B field). The contents of RB54:55 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, VA58:77-b is equal to (RB)56:75-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)52:59.
    - 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.

**TLBELPID = LPIDR_PID**

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**

If RIC=0 or RIC=2, (RB),i:51 (bits i-40:11 of the SET field in (RB)) specify a set of TLB entries, where i is an implementation-dependent value in the range 40:51. Each partition-scoped entry when PRS=0 and either MSR_{HV}=1 or R=0, or each process-scoped entry when PRS=1 in the set is invalidated if any of the following conditions are met for the entry.

- The IS field in RB contains 0b10, or MSR_{HV}=0 and the IS field contains 0b11, and TLBE_{LPID} = LPID_{LPID}.
- The IS field in RB contains 0b01, TLBE_{LPID}=LPID_{LPID}, and TLBEPID=RS_{0:31}.
- The IS field in RB contains 0b11 and MSR_{HV}=1.

How the TLB is divided into the 2^{52-i} sets is implementation-dependent. The relationship of virtual addresses to these sets is also implementation-dependent. However, if, in an implementation, there can be multiple TLB entries for the same virtual address and same partition, then all these entries must be in a single set.

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, all properly-scoped page walk caching associated with partition LPID_{LPID} is invalidated.
- If the IS field in RB contains 0b11 and MSR_{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 RS_{0:31} in partition LPID_{LPID} is invalidated.

If 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 RS_{0:31} in partition LPID_{LPID} is invalidated.
- If the IS field in RB contains 0b10, MSR_{HV}=1, and PRS=0, caching of Partition Tables for partition LPID_{LPID} is invalidated.
- If the IS field in RB contains 0b10 and PRS=1, caching of Process Tables for partition LPID_{LPID} is invalidated.
- If the IS field in RB contains 0b11, MSR_{HV}=1, and PRS=0, all Process Table caching is invalidated.
- If the IS field in RB contains 0b11, MSR_{HV}=1, and PRS=1, all Process Table caching is invalidated.
- If the IS field in RB contains 0b11, MSR_{HV}=0, and PRS=1, caching of Process Tables for partition LPID_{LPID} is invalidated.

When i>40, RB_{40:i} may contain any value and are ignored by the hardware.

**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_{0:39}, RB_{39:62}, RB_{59:63}, RB_{54:55}, and RB_{54:63} are the equivalent of reserved fields, should contain 0s, and are ignored by the hardware. RS_{32:63} 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 `tlbiel` instruction are affected.

MSR_{SF} must be 1 when this instruction is executed; otherwise the results are boundedly undefined.

If the value specified in RS_{0:31}, RB_{54:55}, RB_{56:58}, or RB_{54:51}, 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 5.10, “Translation Table Update Synchronization Requirements” on page 1043 for a description of other requirements associated with the use of this instruction.

**Special Registers Altered:**

None

**Extended Mnemonics:**

Extended mnemonic for `tlbiel`:

<table>
<thead>
<tr>
<th>Extended:</th>
<th>Equivalent to:</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>tlbiel</code></td>
<td><code>tlbiel</code> RB,r0,0,0,0</td>
</tr>
</tbody>
</table>

**Programming Note**

`tlb` and `tlbiel` serve as both basic and extended mnemonics. The Assembler will recognize a `tlb` or `tlbiel` mnemonic with five operands as the basic form, and a `tlb` with two operands or a `tlbiel` mnemonic with one operand as the extended form. In the extended form the RIC, PRS, and R operands, and for `tlbiel` the RS operand, are omitted and assumed to be 0.
The primary use of this instruction by hypervisor software is to invalidate TLB entries prior to reassigning a thread to a new logical partition.

The primary use of this instruction by operating system software is to invalidate TLB entries that were created by the hypervisor using an implementation-specific hypervisor-managed TLB facility, if such a facility is provided.

tlbie may be executed on a given thread even if the sequence tlbie - eieio - tlbsync - ptesync is concurrently being executed on another thread.

See also the Programming Notes with the description of the tlbie instruction.

Programming Note

An operating system that uses HPT translation should only use 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 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.

Programming Note

The tlbie instruction should not be used to synchronize the completion of tlbie.
5.10 Translation Table Update Synchronization Requirements

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 *tlbie* 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 5.7.12.) 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 5.5). 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, or hrfid 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.

### 5.10.1 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 5.9.3, "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.

```plaintext
tlbie instruction(s) specifying the same LPID operand
eltieo tlbisync
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.

```plaintext
eleio tlbisync tlbisync
```

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 `tlbisync` instructions should only be used as part of these sequences.

The corresponding sequence for Segment Table updates uses `slbieg` in place of `tlbie` and `tlbsync` in place of `tlbisync`. 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 `tlbsync` 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 multi-threaded, is sufficient.

--- Programming Note ---

The eieio instruction prevents the reordering of the preceding tlbie, slbieg, or slbiag instructions with respect to the subsequent tlb-sync or slbsync instruction. The tlb-sync 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, slbieg, or slbiag 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 5.10.1.2, “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 tlb-sync or slbsync and the eieio that separates the tlbie or slbieg from the tlb-sync or slbsync can be omitted. In a multi-threaded environment, when tlbie or slbieg is used instead of tlbie or slbieg in a Page or Segment Table update, the synchronization requirements are the same as when tlbie or slbieg 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.

5.10.1.1 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.

```
PTEpp key B ARPN LP key R C WING N pp ^= new values
eieio /* order 1st update before 2nd */
PTEAVA SL H V ^= new values (V=1)
ptesync /* order updates before next Page Table search and before next data access */
```

5.10.1.2 Modifying a Translation Table Entry

General Case (PTE)

If a valid entry is to be modified and the translation instantiated by the entry being modified is to be invalidated, the sequences below can be used to modify the
PTE, 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 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 multithreaded software. If the purpose of the sequence is mainly to collect Reference and Change bit values, the part of the sequence beginning with `tlbie` may be deferred and performed as a bulk invalidation (e.g. for a range of storage or an entire process) after collecting values for a plurality of pages. A similar sequence (i.e. using `Load And Reserve` and `Store Conditional` instructions) can be used to update a Segment Table Entry but will not interact correctly with non-atomic hardware Reference and Change bit updates.

```plaintext
r6←PTEB, L SW 8M R C Alt EAA
r4←addr(pte)

loop:
  lqarx r2,0,r4
if V=0 abort, else /* to interact with locking */
  stqcx r6,0,r4
bne- loop

ptesync /* order update before tlbie and
 before next Page Table search */
  tlbie(old_EA,old_ESID,old_TA,old_PID,old_LPID,old_LPID)
  eieio /* order tlbie before tlbisync */
  tlbisync /* order after tlbie before ptesync */
  ptesync /* complete the sequence, stores ordered */

/* by first ptesync */

The corresponding sequence for non-atomic hardware updates is the following. (The sequence is equivalent to deleting the STE and then adding a new one.) Mutual exclusion with respect to other software threads may be required. The Reference and Change bit values will not be stable until the entire sequence is completed.

```plaintext
PTEV← 0 /* (other fields don't matter)*/
ptesync /* order update before tlbie and
 before next Page Table search */
  tlbie(old_EA,old_ESID,old_TA,old_PID,old_LPID)
  eieio /* order after tlbie before tlbisync */
  tlbisync /* order after tlbie before ptesync */
  ptesync /* complete the sequence, stores ordered */

/* by first ptesync */

Resetting 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 overwritten another change being performed by multithreaded software, so mutual exclusion may be required.

```plaintext
oldR ← PTEB /* get old R */
if oldR = 1 then
  PTEV ← 0 /* (other fields don't matter)*/
  ptesync /* order update before tlbie and
 before next Page Table search */
  tlbie(old_EA,old_ESID,old_TA,old_PID,old_LPID)
  eieio /* order after tlbie before tlbisync */
  tlbisync /* order after tlbie before ptesync */
  ptesync /* complete the sequence, stores ordered */

/* by first ptesync */

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 Reference or Change bit to 1 or to add access authorities, a simpler sequence suffices because the translation hardware will refresh 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.

\[ \text{PTE}_E = \text{SW} \oplus \text{RPN} \oplus \text{C} \oplus \text{EAA} \leftarrow \text{new values } (V=1) \]

\[ \text{ptesync} \quad /\* \text{order update before next Page Table search and before next data access} \* / \]

**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 /* load dwd 0 of PTE */
    if V=0 abort, else*/to interact with locking*/
    r157:60 /* new SW value */ r1 /* replace SW, in r1 */
    stdcx. PTE_dwd_0 /* store dwd 0 of PTE */
    if still reserved (new SW value, other fields unchanged) */
    bne- loop /* loop if lost reservation */
```

A `ldarx/stdcx., lharx/sthcx., 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.

```
STE_ESID,V \leftarrow \text{new values } (V=1) 
ptesync \quad /\* \text{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 */
```

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.
Chapter 6. Interrupts

6.1 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.

6.2 Interrupt Registers

6.2.1 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 6.5 describes which registers are altered by each interrupt.

---

SRR0

| 0 | 62 63 |

---

SRR1

| 0 | 63 |

---

Figure 52. 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). SRR1_{44} 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 6.5.9, “Program Interrupt” on page 1074.

6.2.2 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 6.5 describes which registers are altered by each interrupt.

---

HSRR0

| 0 | 62 63 |

---

HSRR1

| 0 | 63 |

---

Figure 53. 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 6.4.4.

6.2.3 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 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.

6.2.4 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 6.5.2, 6.5.3, 6.5.4, and 6.5.8. 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.

6.2.5 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 6.5.16. 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.

6.2.6 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 6.5.2, 6.5.3, and 6.5.4.

6.2.7 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.

6.2.8 Hypervisor Emulation Instruction Register

The Hypervisor Emulation Instruction Register (HEIR) is a 32-bit register that is set by the Hypervisor Emulation Assistance interrupt; see Section 6.5.18. The image of the instruction that caused the interrupt is loaded into the register.
6.2.9 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 6.5.19 and Section 3.3.2.

![HMER Register](image)

Figure 61. Hypervisor Maintenance Exception Register

The contents of the HMER are as follows:

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 mtspm 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**

An access to the HMER is likely to be very slow. Software should access it sparingly.

6.2.10 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 6.5.19 and Section 3.3.2.

![HMEER Register](image)

Figure 62. Hypervisor Maintenance Exception Enable Register

6.2.11 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 interrupt.
- Access of an SPR using mfspr/mtspr causes a Facility Unavailable interrupt.
- rfbb, rfd, rfsv, hrfid 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 mtspm and mfspr unless the preceding item applies.

The MSR can also make the Transactional Memory facility unavailable in any privilege state, and MMCR0 can make various components of the Performance Monitor unavailable when accessed in problem state. An access to one of these facilities when it is unavailable causes a Facility Unavailable interrupt.

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.

![FSCR Register](image)

Figure 63. Facility Status and Control Register

The contents of the FSCR are specified below.

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:7</td>
<td><strong>Interruption Cause (IC)</strong></td>
</tr>
</tbody>
</table>

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
04 Execution of a BHRB Instruction
05 Access to a Transactional Memory SPR or execution of a Transactional Memory Instruction
06 Reserved
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

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.

8:50 Reserved
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 7 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
HFSCR58:60 are used to control the availability of Transactional Memory, the Performance Monitor, and the BHRB in problem and privileged non-hypervisor states. FSCR58:60 are reserved since the availability of Transactional Memory is controlled by the MSR, and 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.

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.

6.2.12 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 interrupt.
- Access of an SPR using mfspr/mtspr causes a Hypervisor Facility Unavailable interrupt
- rfebb, rfid, rfscv, hrfid 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.

Programming Note
HFSCR58:60 are used to control the availability of Transactional Memory, the Performance Monitor, and the BHRB in problem and privileged non-hypervisor states. FSCR58:60 are reserved since the availability of Transactional Memory is controlled by the MSR, and the availability of the Performance Monitor and BHRB is controlled by MMCR0.
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: Hypervisor Facility Status and Control Register**

<table>
<thead>
<tr>
<th>IC</th>
<th>Facility Control</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
</tr>
</tbody>
</table>

**Programming Note**

Notice that \texttt{rfebb}, \texttt{rfscv}, [\texttt{h}]\texttt{rfid}, and \texttt{mtmsrd} instructions can cause a TM Bad Thing type Program interrupt even when executed in a privileged state in which TM is made unavailable by the HFSCR. Here are two examples. Both assume that HFSCR\textsubscript{TM}=0; the second assumes that HFSCR\textsubscript{EBS}=1.

- An operating system, running with MSR\textsubscript{TS TM} = 0b000 (N0), sets SRR1\textsubscript{29:31} to 0b101 (T1) then executes \texttt{rfid}. The attempted illegal transaction state transition will cause a TM Bad Thing type Program interrupt, despite the fact that TM is made unavailable in privileged non-hypervisor state by the HFSCR.
- An application program, running with MSR\textsubscript{TS TM} = 0b000 (N0), sets BESCSR\textsubscript{TS} to 0b01 (S) then executes \texttt{rfebb}. The attempted illegal transaction state transition will cause a TM Bad Thing type Program interrupt, despite the fact that TM is made unavailable in problem state by the HFSCR.

This anomaly cannot be caused by the PCR.

- \texttt{rfscv}, [\texttt{h}]\texttt{rfid}, and \texttt{mtmsrd} cannot be executed in the privileged state (problem state) in which TM is made unavailable by the PCR.
- \texttt{rfebb} can be executed in the privileged state in which TM is made unavailable by the PCR, but the PCR bit that makes TM unavailable (the v2.06 bit) also makes \texttt{rfebb} unavailable.

Another difference between the HFSCR and the PCR is that PCR\textsubscript{v2.06}=1 prevents a thread from being simultaneously in problem state and in Transactional or Suspended state and HFSCR\textsubscript{TM}=0 does not. However, if the hypervisor always returns to the partition in Non-transactional state when HFSCR\textsubscript{TM}=0, the partition will be unable to enter Transactional or Suspended state.

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.

**Figure 64. Hypervisor Facility Status and Control Register**

The contents of the HFSCR are specified below.

### Programming Note

There is no bit in this register controlling the availability of the \texttt{stop} instruction because the availability of \texttt{stop} in privileged non-hypervisor state is controlled by the PSSCR. See Section 3.2.3.

**Value**

<table>
<thead>
<tr>
<th>0:7</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:2</td>
<td>Interruption Cause (IC)</td>
</tr>
</tbody>
</table>

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 9.4.1 for a definition of groups A and B.)
- 04: Execution of a BHRB Instruction
- 05: Access to a Transactional Memory SPR or execution of a Transactional Memory instruction
- 06: Reserved
- 07: Access to the \texttt{msgsndp} or \texttt{msgclrp} instructions, the TIR or the DPDES Register

All other values are reserved.

**Value**

<table>
<thead>
<tr>
<th>8:63</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>8:52</td>
<td>Facility Enable (FE)</td>
</tr>
</tbody>
</table>

The FE field controls the availability of various facilities in problem and privileged non-hypervisor states as specified below.

- 09: Access to the \texttt{stop} instruction in privileged non-hypervisor state when one or more of the following conditions exist.
  - PSSCR\textsubscript{EC}=1
  - PSSCR\textsubscript{ESL}=1
  - PSSCR\textsubscript{MTL}>PSSCR\textsubscript{PSLL}
  - PSSCR\textsubscript{RL}>PSSCR\textsubscript{PSLL}
- 0A: Access to the \texttt{msgsndp} or \texttt{msgclrp} instructions, the TIR or the DPDES Register

All other values are reserved.

**Value**

<table>
<thead>
<tr>
<th>53</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>\texttt{msgsndp} instructions and SPRs (MSGP)</td>
</tr>
</tbody>
</table>

- 0: The \texttt{msgsndp} and \texttt{msgclrp} instructions and the TIR and DPDES registers are not available in privileged non-hypervisor state.
The `msgsndp` and `msgclrp` instructions and the TIR and DPDES registers are available in privileged non-hypervisor state unless made unavailable by another register.

54 Reserved

55 **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.

56 **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.

57 Reserved

58 **Transactional Memory Facility** (TM)

0 The Transactional Memory Facility SPRs and instructions are not available in problem and privileged non-hypervisor states.

1 The Transactional Memory Facility SPRs and instructions are available in problem and privileged states unless made unavailable by another register.

59 **BHRB Instructions** (BHRB)

0 The BHRB instructions (`clrbhrb`, `mfbhrbe`) are not available in problem and privileged non-hypervisor states.

1 The BHRB instructions (`clrbhrb`, `mfbhrbe`) are available in problem and privileged states unless made unavailable by another register.

60 **Performance Monitor Facility SPRs** (PM)

0 Read and write operations of Performance Monitor SPRs in group A and read operations of Performance Monitor SPRs in group B are not available in problem and privileged non-hypervisor states; read and write operations to privileged Performance Monitor registers (SPRs 784-792, 795-798) are not available in privileged non-hypervisor state. (See Section 9.4.1 for a definition of groups A and B.) Performance Monitor exceptions do not cause Performance Monitor interrupts to occur when the thread is in problem or privileged states.

1 Read and write operations of Performance Monitor SPRs in group A and read operations of Performance Monitor SPRs in group B are available in problem and privileged states unless made unavailable by another register; read and write operations to privileged Performance Monitor registers (SPRs 784-792, 795-798) are available in privileged state; Performance Monitor interrupts to occur if $\text{MSR}_{\text{EE}}=1$ and $\text{MMCR0}_{\text{EBE}}=0$. See Section 9.2 of Book III for additional information

61 **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.

62 **Vector and VSX Facilities** (VECVSX)

0 The facilities whose availability is controlled by either MSRVEC or MSRVSX are not available in problem and privileged non-hypervisor states.

1 The facilities whose availability is controlled by either MSRVEC or MSRVSX are available in problem and privileged states unless made unavailable by another register.

63 **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.
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 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.
The following tables summarize the interrupts that occur as a result of accessing the non-privileged Performance Monitor registers in problem state when MMCR0 PMCC, 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>PMCC</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</td>
<td>769</td>
<td>HU</td>
<td>FU</td>
<td>HU</td>
<td>HU</td>
<td>HU</td>
<td>HE</td>
<td>HU</td>
<td>HU</td>
</tr>
<tr>
<td>MMCRA</td>
<td>770</td>
<td>HU</td>
<td>FU</td>
<td>HU</td>
<td>HU</td>
<td>HU</td>
<td>HE</td>
<td>HU</td>
<td>HU</td>
</tr>
<tr>
<td>PMC1</td>
<td>771</td>
<td>HU</td>
<td>FU</td>
<td>HU</td>
<td>HU</td>
<td>HU</td>
<td>HE</td>
<td>HU</td>
<td>HU</td>
</tr>
<tr>
<td>PMC2</td>
<td>772</td>
<td>HU</td>
<td>FU</td>
<td>HU</td>
<td>HU</td>
<td>HU</td>
<td>HE</td>
<td>HU</td>
<td>HU</td>
</tr>
<tr>
<td>PMC3</td>
<td>773</td>
<td>HU</td>
<td>FU</td>
<td>HU</td>
<td>HU</td>
<td>HU</td>
<td>HE</td>
<td>HU</td>
<td>HU</td>
</tr>
<tr>
<td>PMC4</td>
<td>774</td>
<td>HU</td>
<td>FU</td>
<td>HU</td>
<td>HU</td>
<td>HU</td>
<td>HE</td>
<td>HU</td>
<td>HU</td>
</tr>
<tr>
<td>PMC5</td>
<td>775</td>
<td>HU</td>
<td>FU</td>
<td>HU</td>
<td>HU</td>
<td>HU</td>
<td>HE</td>
<td>HU</td>
<td>HU</td>
</tr>
<tr>
<td>PMC6</td>
<td>776</td>
<td>HU</td>
<td>FU</td>
<td>HU</td>
<td>HU</td>
<td>HU</td>
<td>HE</td>
<td>HU</td>
<td>HU</td>
</tr>
<tr>
<td>MMCR0</td>
<td>779</td>
<td>HU</td>
<td>FU</td>
<td>HU</td>
<td>HU</td>
<td>HU</td>
<td>HE</td>
<td>HU</td>
<td>HU</td>
</tr>
<tr>
<td>SIER</td>
<td>768</td>
<td>HU</td>
<td>FU</td>
<td>HU</td>
<td>HU</td>
<td>HU</td>
<td>See</td>
<td>See</td>
<td>See</td>
</tr>
<tr>
<td>SIAR</td>
<td>780</td>
<td>HU</td>
<td>FU</td>
<td>HU</td>
<td>HU</td>
<td>HU</td>
<td>See</td>
<td>See</td>
<td>See</td>
</tr>
<tr>
<td>SDAR</td>
<td>781</td>
<td>HU</td>
<td>FU</td>
<td>HU</td>
<td>HU</td>
<td>HU</td>
<td>See</td>
<td>See</td>
<td>See</td>
</tr>
<tr>
<td>MMCR1</td>
<td>782</td>
<td>HU</td>
<td>FU</td>
<td>HU</td>
<td>HU</td>
<td>HU</td>
<td>See</td>
<td>See</td>
<td>See</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 mtspr instruction description in Section 4.4.4 for additional information.) FU or HU interrupts do not occur regardless of the value of MMCR0 PMCC or HFSCR PM.

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 PMCC or HFSCR PM.

4. An HU interrupt occurs if HFSCR PM=0 when this SPR is accessed in either problem state or privileged non-hypervisor state.

---

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).
6.3 Interrupt Synchronization

When an interrupt occurs, in general SRR0 or HSRR0 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 or HSRR0 may or may not have completed execution, depending on the interrupt type. The only exception is that if an mtspr sequence started by mtgsr is active when the interrupt occurs, some of the sequence's mtsprs beyond the instruction pointed to by SRR0 or HSRR0 may have been executed; see Chapter 11.

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.

6.4 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
- Performance Monitor

External, Decrementer, Hypervisor Decrementer, Directed Privileged Doorbell, Directed Hypervisor 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.

6.4.1 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 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, except that if an mtspr sequence started by mtgsr is active when the interrupt occurs, some of the sequence's mtsprs beyond the interrupt point may have been executed; see Chapter 11 of Book III.

6.4.2 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 6.5.9, “Program Interrupt” on page 1074.
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 6.5.9.
4. No instruction following the instruction addressed by SRR0 appears to have begun execution, except that if an mtspr sequence started by mtgsr is active when the interrupt occurs, some of the sequence's mtsprs beyond the interrupt point may have been executed; see Chapter 11.
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.
6.4.3 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 6.5.2) and System Call Vectored interrupts (see Section 6.5.27) differs from the description given below in several respects.

1. SRR0 or HSRR0 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 or HSRR1 are loaded with information specific to the interrupt type.
3. Bits 0:32, 37:41, and 48:63 of SRR1 or HSRR1 are loaded with a copy of the corresponding bits of the MSR.
4. The MSR is set as shown in Figure 65 on page 1064. In particular, MSR bits IR and DR are set as specified by LPCR_AIL (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 66 on page 1065. An offset may be applied to get the effective addresses, as specified by LPCR_AIL (see Section 2.2).

Interrupts do not clear reservations obtained with lbarx, lharx, lwarx, ldarx, 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`, `stcx`, `stwcx`, `stdcx`, or `stqcx`, to clear the reservation if one is outstanding, to ensure that a lbarx, lharx, lwarx, ldarx, or lqarx in the interrupted program is not paired with a stbcx, stcx, stwcx, stdcx, or stqcx on the “new” program.
- “eieio, tlbsync, slbsync, ptiesync,” 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 sync instruction suffices.)
- `isync` or `rfid`, to ensure that the instructions in the “new” program execute in the “new” context.
- `treclaim`, to ensure that any previous use of the transactional facility is terminated.
- `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 `l mw` 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 `lw ax` 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 `lw z` 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 `lw z` 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.)

If a program modifies an instruction that it or another program will subsequently execute and the execution of the instruction causes an interrupt, the state of storage and the content of some registers may appear to be inconsistent to the interrupt handler program. For example, this could be the result of one program executing an instruction that causes a Hypervisor Emulation Assistance interrupt just before another instance of the same program stores an Add Immediate instruction in that storage location. To the interrupt handler code, it would appear that a hardware generated the interrupt as the result of executing a valid instruction.
6.4.4 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]`, `bclr[l]`, `bcctr[l]`

2. **Fixed-Point Load and Store Instructions**
   - `lbz`, `lbzx`, `lhz`, `lhzx`, `lwz`, `lwzx`, `ld`, `ldx`, `stb`, `stbx`, `sth`, `sthx`, `stw`, `stwx`, `std`, `stdx`

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}=0b10.

3. **Arithmetic instructions**
   - `addi`, `addis`, `add`, `subf`, `neg`

4. **Compare instructions**
   - `cmpi`, `cmp`, `cmpli`, `cmpl`

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`, `rldcl`, `rldcr`, `rlwnm`, `rldimi`, `rlwimi`, `sld`, `slw`, `srd`, `srw`

7. **Other instructions**
   - `isync`
   - `rfid`, `hrfid`
   - `mtspr`, `mfspr`, `mtmsrd`, `mfmsr`

---

**Programming Note**

Hardware reports system integrity problems via Machine Check and System Reset interrupts that set SRR1_{62} to 0. All other interrupts that set the SRRs, including Machine Check and System Reset interrupts that do not themselves report integrity problems, copy MSR_{RI} to SRR1_{62}. (All interrupts that set the SRRs set MSR_{RI} to 0.) To interact correctly with this behavior, interrupt handlers for interrupts that set the SRRs should do as follows.

- In each such interrupt handler, interpret SRR1_{62} as:
  - 0: interrupt is not recoverable
  - 1: interrupt is recoverable

- In each such interrupt handler, when enough state has been saved that another interrupt that sets the SRRs can be recovered from, set MSR_{RI} to 1.

- In each such interrupt handler, do the following (in order) just before returning:
  1. Set MSR_{RI} to 0.
  2. Set SRR0 and SRR1 to the values to be used by `rfid`. The new value of SRR1 should have bit 62 set to 1 (which will happen naturally if SRR1 is restored to the value saved there by the interrupt, because the interrupt handler will not be executing this sequence unless the interrupt is recoverable).
  3. Execute `rfid`.

---

**Programming Note**

Because interrupts that set the HSRRs preserve MSR_{RI} instead of setting it to 0 as is done by interrupts that set the SRRs, handlers for interrupts that set the HSRRs must prevent additional such interrupts from occurring until enough state has been saved that another such interrupt can be recovered from, and also when the HSRRs have been restored prior to executing `hrfid`. Required behavior during those intervals includes the following.

- Keep MSR_{HV EE PR}=0b100. (This state prevents many such interrupts from occurring.)
- Execute only defined instructions that are not in invalid form.
- Pin the first page of the hypervisor’s Process Table
- Ensure that the PTE mapping the first page of the hypervisor’s Process Table has the Reference bit set and has no other reason to cause an exception.
Similarly, fetching instructions may have the side effect of altering the contents of HSRR0 and HSRR1 unless \( \text{MSR}_{\text{HV IR}} = 0b10 \).

---

**Programming Note**

Instructions excluded from the list include the following.

- Instructions that set or use \( \text{XER}_{\text{CA}} \)
- Instructions that set \( \text{XER}_{\text{OV}} \) or \( \text{XER}_{\text{SO}} \)
- \( \text{andi} \), \( \text{andis} \), and fixed-point instructions with \( \text{Rc}=1 \) (Fixed-point instructions with \( \text{Rc}=1 \) can be replaced by the corresponding instruction with \( \text{Rc}=0 \) followed by a \text{Compare} instruction.)
- All floating-point instructions
- \( \text{mftb} \)

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.)
6.5 Interrupt Definitions

Figure 65 shows all the types of interrupts and the values assigned to the MSR for each. Figure 66 shows the effective address of the interrupt vector for each interrupt type. (Section 5.7.5 on page 987 summarizes all architecturally defined uses of effective addresses, including those implied by Figure 66.)

<table>
<thead>
<tr>
<th>Interrupt Type</th>
<th>MSR Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>IR DR FE0 FE1 EE RI ME HV</td>
</tr>
<tr>
<td>System Reset</td>
<td>0 0 0 0 0 0 0 p 1</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</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>System Call Vectored</td>
<td>r r 0 0 0 - - - -</td>
</tr>
</tbody>
</table>
Figure 65. MSR setting due to interrupt

<table>
<thead>
<tr>
<th>Interrupt Type</th>
<th>MSR Bit</th>
<th>IR DR FE0 FE1 EE RI ME HV</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>bit is set to 0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>bit is set to 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>bit is not altered</td>
<td></td>
</tr>
</tbody>
</table>

r  for interrupts for which LPCRAIL applies, if LPCRAIL = 2 or 3, set to 1; otherwise set to 0
p  if the interrupt occurred while the thread was in power-saving mode, set to 1; otherwise not altered
e  if LPES = 0, set to 1; otherwise not altered
h  if LPES = 1, set to 0; otherwise not altered
s  if LEV = 1, set to 1; otherwise not altered

*Settings for Other Bits*

Bits bit 5, TM, VEC, VSX, PR, FP, and PMM are set to 0.

The TE field is set to 0b00.

TM, FP, VEC, VSX, and bit 5 are set to 0.

If the interrupt results in HV being equal to 1, 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.

If the TS field contained 0b10 (Transactional) when the interrupt occurred, the TS field is set to 0b01 (Suspended); otherwise the TS field is not altered.

Reserved bits are set as if written as 0.
## 6.5.1 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 PSSCREG=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
- Hypervisor Maintenance

When address translation is disabled, use of any of the effective addresses that are shown as reserved in Figure 66 risks incompatibility with future implementations.

### Programming Note

When address translation is disabled, use of any of the effective addresses that are shown as reserved in Figure 66 risks incompatibility with future implementations.

### Figure 66. Effective Address of Interrupt Vectors by Interrupt Type

<table>
<thead>
<tr>
<th>Effective Address</th>
<th>Interrupt Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>00_000_0100</td>
<td>System Reset</td>
</tr>
<tr>
<td>00_000_0200</td>
<td>Machine Check</td>
</tr>
<tr>
<td>00_000_0300</td>
<td>Data Storage</td>
</tr>
<tr>
<td>00_000_0380</td>
<td>Data Segment</td>
</tr>
<tr>
<td>00_000_0400</td>
<td>Instruction Storage</td>
</tr>
<tr>
<td>00_000_0480</td>
<td>Instruction Segment</td>
</tr>
<tr>
<td>00_000_0500</td>
<td>External</td>
</tr>
<tr>
<td>00_000_0600</td>
<td>Alignment</td>
</tr>
<tr>
<td>00_000_0700</td>
<td>Program</td>
</tr>
<tr>
<td>00_000_0800</td>
<td>Floating-Point Unavailable</td>
</tr>
<tr>
<td>00_000_0900</td>
<td>Decrementer</td>
</tr>
<tr>
<td>00_000_0980</td>
<td>Hypervisor Decrementer</td>
</tr>
<tr>
<td>00_000_0a00</td>
<td>Directed Privileged Doorbell</td>
</tr>
<tr>
<td>00_000_0b00</td>
<td>Reserved</td>
</tr>
<tr>
<td>00_000_0c00</td>
<td>System Call</td>
</tr>
<tr>
<td>00_000_0d00</td>
<td>Trace</td>
</tr>
<tr>
<td>00_000_0e00</td>
<td>Hypervisor Data Storage</td>
</tr>
<tr>
<td>00_000_0e20</td>
<td>Hypervisor Instruction Storage</td>
</tr>
<tr>
<td>00_000_0e40</td>
<td>Hypervisor Emulation Assistance</td>
</tr>
<tr>
<td>00_000_0e60</td>
<td>Hypervisor Maintenance</td>
</tr>
<tr>
<td>00_000_0e80</td>
<td>Directed Hypervisor Doorbell</td>
</tr>
<tr>
<td>00_000_0e80</td>
<td>Hypervisor Virtualization</td>
</tr>
<tr>
<td>00_000_0ea0</td>
<td>Reserved</td>
</tr>
<tr>
<td>00_000_0ec0</td>
<td>Reserved for implementation-dependent interrupt for performance monitoring</td>
</tr>
<tr>
<td>00_000_0f00</td>
<td>Performance Monitor</td>
</tr>
<tr>
<td>00_000_0f20</td>
<td>Vector Unavailable</td>
</tr>
<tr>
<td>00_000_0f40</td>
<td>VSX Unavailable</td>
</tr>
<tr>
<td>00_000_0f60</td>
<td>Facility Unavailable</td>
</tr>
<tr>
<td>00_000_0f80</td>
<td>Hypervisor Facility Unavailable</td>
</tr>
<tr>
<td>00_000_0fa0</td>
<td>Reserved</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>00_000_0ffe</td>
<td>Reserved</td>
</tr>
<tr>
<td>00_0001_7000</td>
<td>System Call Vectored</td>
</tr>
<tr>
<td>00_0001_7020</td>
<td>System Call Vectored</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>00_0001_7fe0</td>
<td>System Call Vectored</td>
</tr>
<tr>
<td>00_0001_7fff</td>
<td>(end of scv interrupt vectors)</td>
</tr>
</tbody>
</table>

1 The values in the Effective Address column are interpreted as follows.

- 00_0000_0nnn means 0x0000_0000_0000_0nnn unless the values of LPCRAIL and MSRHVIRDR cause the application of an effective address offset. See the description of LPCRAIL in Section 2.2 for more details.

- 0...0_0001_7nnn means 0x0000_0000_0001_7nnn unless the values of LPCRAIL and MSRHVIRDR cause the usage of an alternate effective address. See the description of LPCRAIL in Section 2.2 for details.

2 Effective addresses 0x0000_0000_0000_0000 through 0x0000_0000_0000_00FF are used by software and will not be assigned as interrupt vectors.
Hypervisor Virtualization exception

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 PSSCREEC 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 MSR[EE]=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] Exception</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>Reserved</td>
</tr>
<tr>
<td>0001</td>
<td>Reserved</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:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>The interrupt did not occur when the thread was in power-saving mode.</td>
</tr>
<tr>
<td>01</td>
<td>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.</td>
</tr>
</tbody>
</table>

1066 Power ISA™ III
If the interrupt did not occur while 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 SRR146:47, to determine what state to restore. (To avoid implementation-dependence, the hypervisor must assume that only the resources indicated in SRR146:47 have been preserved.

If the interrupt did not occur while the thread was in a power-saving level that was entered when PSSCR\textsubscript{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\textsubscript{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 65 on page 1064.

In addition, if the interrupt occurs when the thread is in a power-saving level that was entered when PSSCR\textsubscript{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.

6.5.2 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 5.6), or by an error in the storage subsystem.

When the thread is not in power-saving mode, Machine Check interrupts are enabled when MSR\textsubscript{ME}=1; if MSR\textsubscript{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\textsubscript{S1}=1 and cannot occur when LPCR\textsubscript{S1}=0. When the thread is in a power-saving level that allows loss of hypervisor state, Machine Check interrupts are treated as enabled LPCR\textsubscript{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 5.6), 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:
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 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.

SRR0

If the interrupt did not occur 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.

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

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.

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

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.

Programming Note

If the interrupt did not occur while the thread was in a power-saving level that was entered when PSSCRREC=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 PSSCRREC=1, set to 1 if the thread is in a recoverable state; otherwise set to 0.

62 If the interrupt did not occur while the thread was in a power-saving level that was entered when PSSCRREC=1, loaded from bit 62 of the MSR if the thread is in a recoverable state; otherwise set to 0.

Others

Set to an implementation-dependent value.

MSR

See Figure 65.

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 5.7.8.2 and 5.7.9.2.) 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.
6.5.3 Data Storage Interrupt

A Data Storage interrupt occurs when no higher priority exception exists and either
(a) a copy-paste transfer other than from main storage to a properly initiated accelerator is attempted, or
(b) (MSRHV PR=0b10) & (MSRDR=0), or
(c) HPT translation is being performed, the value of the expression
   \((\text{MSRHV PR}=0b10) \land (\neg \text{VPM} \land \neg \text{PRTEV}) \land \text{MSRDR})\)
   is 1, and a data access cannot be performed, except for the case of MSRHV PR=0b10,
   VPM=0, LPCRKBV=1, and a Virtual Storage Page Class Key Protection exception exists or
(d) 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, or process-scoped translation either does not complete or prevents the data access from being performed

for any of the following reasons that can occur in the respective translation state. (In the expression for (a) above, "\(\neg \text{PRTEV}\)" is shorthand representing the case of an invalid segment table descriptor stopping the translation process.)

- Data address translation is enabled (MSRPR=1) and the effective or virtual address of any byte of the storage location specified by a Load, Store, icbi, dcbbz, dbcbz, dcbfi, or dcbf[i] 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, stq, lwat, ldat, lbax, lbarx, lwarx, ldarx, lqarx, stwat, stdat, stbcx, sthcx, stwcx, stdcx, 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, normal memory is specified at the target of a paste 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 LPCRKBV=0.
- The process- and partition-scoped page attributes conflict.
- An unsupported radix tree configuration is found in the process-scoped tables.
- 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 MSRDR=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, stdcx, or stqcx 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:

<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.</td>
</tr>
<tr>
<td>Others</td>
<td>Loaded from the MSR.</td>
</tr>
<tr>
<td>MSR</td>
<td>See Figure 65.</td>
</tr>
<tr>
<td>DSISR</td>
<td>Set to 0.</td>
</tr>
</tbody>
</table>
Set to 1 if MSRDR=1 and the translation for an attempted access is not found in the Page Table; otherwise set to 0.

Set to 1 if the process- and partition-scoped page attributes conflict; otherwise set to 0.

Set to 1 if the access is not permitted by Figure 44, or the privilege, read, or read/write bits in Figure 45 as appropriate; otherwise set to 0.

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.

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.

Set to 0.

DAR 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 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 DSISR60=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 quadword for which access was attempted in the page that caused the exception, for a quadword Load or Store instruction (i.e., a Load or Store instruction for which the storage operand is a quadword; “first” refers to address order; see Section 6.7)
    - a byte in the first aligned double-word for which access was attempted in the page that caused the exception, for a non-quadword Load or Store instruction
  - set as described in the previous major bullet, except that the low order 5 bits are undefined, for a Data Address Watchpoint match

---

**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 5.7.12.

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.

Set to 0.

Set to 1 if an accelerator is specified as the source of a copy instruction, normal memory is specified as the target of a paste 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.

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.
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 DSI-causing 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 66.

### 6.5.4 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, ica bi, dcbz, dcbst, or dcbf[l] instruction cannot be translated to a virtual address.

For Radix Tree Translation (in other than hypervisor real mode), 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, ica bi, dcbz, dcbst, or dcbf[l] instruction, EA0:1 = 0b01 or EA 0:1 = 0b10 when MSR HVPR 0b10 and data address translation is enabled, or EA2:63 is outside the range translated by the appropriate Radix Tree.

If a stbcx., sthcx., 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:

<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>SRR1:36</td>
<td>Set to 0.</td>
</tr>
<tr>
<td>SRR1:47</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 65.</td>
</tr>
<tr>
<td>DSISR</td>
<td>Set to an undefined value.</td>
</tr>
</tbody>
</table>

### Programming Note

A Data Segment interrupt occurs if MSRPR=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).

### 6.5.5 Instruction Storage Interrupt

An Instruction Storage interrupt occurs when no higher priority exception exists and either

(a) HPT Translation is being performed, the value of the expression

\[ (MSRHPR=0b10) \land (\neg (\neg VPM \land \neg PRTEV) \land MSRIR) \]

is 1, and the next instruction to be executed cannot be fetched, or

(b) Radix Tree translation is being performed and process-scoped translation prevents the next instruction to be executed from being fetched for any of the following reasons. (In the expression for (a) above, “\( \neg PRTEV \)” is shorthand representing the case of an invalid segment table descriptor stopping the translation process.)

- 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 fetch access violates storage protection.

The process- and partition-scoped page attributes conflict.

An unsupported radix tree configuration is found in the process-scoped tables.

A reference bit update cannot be performed in a process-scoped PTE.

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**
- Set to 1 if MSRIR=1 and the translation for an attempted access is not found in the Page Table; otherwise set to 0.
- Set to 1 if the process- and partition-scoped page attributes conflict; otherwise set to 0.
- 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; otherwise set to 0.
- Set to 1 if the access is not permitted by Figure 44 or 46, as appropriate; otherwise set to 0.
- Set to 1 if the access is not permitted by virtual page class key protection; otherwise set to 0.
- Set to 0.
- Set to 1 if an unsupported radix tree configuration is found during the translation process; otherwise set to 0.

**Program 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 5.7.12.

**Others**
Loaded from the MSR.

**MSR**
See Figure 65.

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 66.

### 6.5.6 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.

For Radix Tree Translation (in other than hypervisor real mode), an Instruction Segment interrupt occurs when no higher priority exception exists and the next instruction to be executed cannot be fetched because EA0:1=0b01 or EA0:1=0b10 when MSRHVPR ≠ 0b10 and instruction address translation is enabled, or EA2:63 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**
- Set to 0.
- Set to 0.

**Others**
Loaded from the MSR.

**MSR**
See Figure 65 on page 1064.

Execution resumes at effective address 0x0000_0000_0000_0480, possibly offset as specified in Figure 66.

**Program Note**

An Instruction Segment interrupt occurs if MSRIR=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).
6.5.7 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.

6.5.7.1 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

\[
\text{MSREE} \& \neg (\text{MSRHV} \& \neg \text{MSRPR} \& \text{LPCRHEIC}) | \neg (\text{LPES} \& \neg (\text{MSRHV} \| \text{MSRPR}))
\]

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 MSREE=1. This enables the Hypervisor Interrupt Virtualization 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 65 on page 1064.

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 65 on page 1064.

Execution resumes at effective address 0x0000_0000_0000_0500, possibly offset as specified in Figure 66.

6.5.7.2 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

\[
\text{MSREE} \& \neg (\text{MSRHV} \| \text{MSRPR})
\]

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**
    - 33:36 Set to 0.
    - 42:47 Set to 0.
    - Others Loaded from the MSR.
  - **MSR** See Figure 65 on page 1064.

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 65 on page 1064.

Execution resumes at effective address 0x0000_0000_0000_0500, possibly offset as specified in Figure 66.

6.5.8 Alignment Interrupt

Many causes of Alignment interrupt involve storage operand alignment. Storage operand alignment is defined in Section 1.11.1 of Book I.
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. 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, lwat, ldat, lharx, lwaxr, ldaxr, lqarx, stwat, stdat, sthcx, stwcx, stdcx, 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.

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 ldq, stfdp, stfdpx, lxsibhx, or stxsihx is unaligned.
- The storage operand of lq or stq is unaligned.
- 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, ldq, stfdp, stfdpx, 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  
33:36  Set to 0.
42:47  Set to 0.
Others  Loaded from the MSR.

MSR   See Figure 65.

### DAR
Set to the effective address computed by the instruction, except that 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 66.

#### 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.

### 6.5.9 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

\[(\text{MSR}_{\text{FEO}} \mid \text{MSR}_{\text{FEI}}) \& \text{FPSCR}_{\text{FEX}}\]

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.

#### TM Bad Thing
A TM Bad Thing type Program interrupt is generated when any of the following occurs.

- An rfebb, rfid, rfscv, hrfdi, or mtmsrd instruction attempts to cause an illegal transaction state transition (see Section 3.2.2).

- An rfid, rfscv, hrfd, or mtmsrd instruction, executed when TM is made unavailable in problem state by the PCR (PCRv2.06=1), attempts to cause a transition to problem state and also a transaction state transition that Table 3 on page 947 shows as legal and as resulting in the thread being in Transactional or Suspended state.
An attempt is made to execute `trechkpt` in Transactional or Suspended state or when TEXASRF<sub>F3</sub>=0.

An attempt is made to execute `tend` in Suspended state.

An attempt is made to execute `treclaim` in Non-transactional state.

An attempt is made to execute an `mtspr` instruction targeting a TM register in other than Non-transactional state, with the exception of TFHAR in Suspended state.

An attempt is made to execute a `stop` instruction in Suspended state.

### Privileged Instruction

The following applies if the instruction is executed when MSR<sub>PR</sub> = 1.

A Privileged Instruction type Program interrupt is generated when execution is attempted of a privileged instruction, or of an `mtspr` or `mfspr` instruction with an SPR field that contains a value having spr<sub>0</sub>=1.

The following applies if the instruction is executed when MSR<sub>HV</sub><sub>PR</sub> = 0b00 and LPCREVIRT=0.

A Privileged Instruction type Program interrupt is generated when execution is attempted of an `mtspr` or `mfspr` instruction with an SPR field that designates an SPR that is accessible by the instruction only when the thread is in hypervisor state, or when execution of a hypervisor-privileged instruction is attempted.

#### Programming Note

These are the only cases in which a Privileged Instruction type Program interrupt can be generated when MSR<sub>PR</sub>=0. They can be distinguished from other causes of Privileged Instruction type Program interrupts by examining SRR<sub>1</sub> (the bit in which MSR<sub>PR</sub> 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<sub>FE0 FE1</sub> = 0b00, FPSCR<sub>FEX</sub> = 1, and an instruction is executed that changes MSR<sub>FE0 FE1</sub> 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<sub>FE0 FE1</sub> are context synchronizing, and therefore are not initiated until all preceding instructions have reported all exceptions they will cause.

- If MSR<sub>FE0 FE</sub> = 0b11, set to the effective address of the instruction that caused the Floating-Point Enabled Exception.
- If MSR<sub>FE0 FE</sub> = 0b01 or 0b10, set to the effective address of the first instruction that caused a Floating-Point Enabled Exception since the most recent time FPSCR<sub>FEX</sub> 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.)

**SRR1**

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
<th>Exception Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>33:36</td>
<td>Set to 0.</td>
<td></td>
</tr>
<tr>
<td>42</td>
<td>Set to 1 for a TM Bad Thing type Program interrupt; otherwise set to 0.</td>
<td></td>
</tr>
<tr>
<td>43</td>
<td>Set to 1 for a Floating-Point Enabled Exception type Program interrupt; otherwise set to 0.</td>
<td></td>
</tr>
<tr>
<td>44</td>
<td>Set to 0.</td>
<td></td>
</tr>
<tr>
<td>45</td>
<td>Set to 1 for a Privileged Instruction type Program interrupt; otherwise set to 0.</td>
<td></td>
</tr>
<tr>
<td>46</td>
<td>Set to 1 for a Trap type Program interrupt; otherwise set to 0.</td>
<td></td>
</tr>
<tr>
<td>47</td>
<td>Set to 0 if SRR0 contains the address of the instruction causing the exception and there is only one such instruction; otherwise set to 1.</td>
<td></td>
</tr>
</tbody>
</table>
6.5.10 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 MSRFP=0.

The following registers are set:

- **SRR0** Set to the effective address of the instruction that caused the interrupt.
- **SRR1**
  - 33:36 Set to 0.
  - 42:47 Set to 0.
- **Others** Loaded from the MSR.

MSR See Figure 65 on page 1064.

Execution resumes at effective address 0x0000_0000_0000_0700, possibly offset as specified in Figure 66.

6.5.11 Decrementer Interrupt

A Decrementer interrupt occurs when no higher priority exception exists, a Decrementer exception exists, and MSRFE=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 65 on page 1064.

Execution resumes at effective address 0x0000_0000_0000_0900, possibly offset as specified in Figure 66.

### 6.5.12 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.

$$(\text{MSR}_{EE} | \neg (\text{MSR}_{HV}) | \text{MSR}_{PR}) & \text{HDICE}$$

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 65 on page 1064.

Execution resumes at effective address 0x0000_0000_0000_0980, possibly offset as specified in Figure 66.

---

**Programming Note**

Because the value of MSR$_{EE}$ is always 1 when the thread is in problem state, the simpler expression

$$(\text{MSR}_{EE} | \neg (\text{MSR}_{HV})) & \text{HDICE}$$

is equivalent to the expression given above.

### 6.5.13 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 10) are received and accepted by the thread.

---

**Programming Note**

An attempt to execute an sc instruction with LEV=1 in problem state should be treated as a programming error.

### 6.5.14 System Call Interrupt

A System Call interrupt occurs when a System Call instruction is executed.

The following registers are set:

**SRR0**
Set to the effective address of the instruction following the System Call instruction.

**SRR1**
- 33:36 Set to 0.
- 42:47 Set to 0.
**Others**
Loaded from the MSR.

**MSR**
See Figure 65 on page 1064.

Execution resumes at effective address 0x0000_0000_0000_0C00, possibly offset as specified in Figure 66.

### 6.5.15 Trace Interrupt

A Trace interrupt occurs when no higher priority exception exists and any instruction except rfid, hrfd, rfsrv, or a Power-Saving Mode instruction is successfully completed, provided any of the following is true:

- the instruction is mtmsr[d] and MSR$_{TE}$=0b10 when the instruction was initiated,
- the instruction is not mtmsr[d] and MSR$_{TE}$=0b10,
- the instruction is a Branch instruction and MSR$_{TE}$=0b01, or
- a CIABR match occurs.

Successful completion for an instruction means that the instruction caused no other interrupt and, if the thread
is in Transactional state, did not cause the transaction to fail in such a way that the instruction did not complete (see Section 5.3.1 of Book II). Thus a Trace interrupt never occurs for a System Call or System Call Vectored instruction, or for a Trap instruction that traps, or for a dcbo that is executed in Transactional state. 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**
- Set to 1.
- Set to 0.
- Set to 1 if the traced instruction 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).
- Set to 1 if 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).
- Set to 1 if the traced instruction is the result of a CIABR match.
- Set to 0.
- Others: Loaded from the MSR.

**Programming Note**

Bit 33 is set to 1 for historical reasons.

**SIAR**
For all Trace interrupts other than those caused by a CIABR match, set to the effective address of the traced instruction; 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; 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 MMCRRPMAE=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 65 on page 1064.

Execution resumes at effective address 0x0000_0000_0000_00D0, possibly offset as specified in Figure 66. For a Trace interrupt resulting from execution of an instruction that modifies the value of MSRIR, MSRDR, MSRHV, or LPCRAIL, the Trace interrupt vector location is based on the modified values.

### 6.5.16 Hypervisor Data Storage Interrupt

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

1. (a) HPT translation is being performed, VPM=0, LPCRKBV=1, and a Virtual Storage Page Class Key Protection exception exists or
2. (b) HPT translation is being performed, the value of the expression (¬MSRDR) | (VPM & PRTEV & MSRDR) is 1, and a data access cannot be performed, or
3. (c) Radix Tree translation is being performed and partition-scoped translation either does not complete or prevents an access from being performed for any of the following reasons that can occur in the respective translation state. (In the expression for (b) above, “PRTEV” is shorthand indicating that an invalid segment table descriptor did not stop the translation process. Note that an SLB hit may satisfy this condition even when the Process Table Entry is invalid.)

- HR=0, data address translation is enabled (MSRDR=1) and the virtual address of any byte of
the storage location specified by a Load, Store, 
icbi, dcbz, dcbst, or dcbf[l] 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 Load, Store, 
icbi, dcbz, dcbst, or dcbf[l] instruction cannot be trans-
lated 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 invalid radix tree configuration found 
in the partition-scoped tables, this type of exception 
will also be reported outside of hypervisor real 
mode for translation mode mismatches including 
UPRT=0 when HR=1, LPID=0 if MSR\textsubscript{hyp}=0 when 
HR=1, and HR=0 for LPID=0 when HR=1 for 
another partition ID.

A reference or change bit update in a parti-
tion-scoped PTE cannot be performed (including for the process-scoped PDE or PTE or process 
table entry for a radix guest.

**Programming Note**

> 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{or}=0), and the virtual address of any byte of the 
storage location specified by a Load, Store, 
icbi, dcbz, dcbst, or dcbf[l] instruction cannot be 
translated to a real address by means of the virtual 
real addressing mechanism.

The effective address specified by a \texttt{icq}, \texttt{stq}, \texttt{lwat}, 
\texttt{ldat}, \texttt{ldarx}, \texttt{lharx}, \texttt{lwax}, \texttt{ldarx}, \texttt{iqaxr}, \texttt{stwat}, 
\texttt{stdat}, \texttt{stdcx}, \texttt{stdhx}, \texttt{stdwx}, \texttt{stdcx}, or \texttt{stqcx} 
instruction refers to storage that is Write Through 
Required or Caching Inhibited; or the effective 
address specified by a \texttt{copy} or \texttt{paste} instruction refers 
to storage that is Caching Inhibited; or the 
effective address specified by a \texttt{lwat}, \texttt{ldat}, \texttt{stwat}, 
or \texttt{stdat} instruction refers to storage that is 
Guarded.

An accelerator is specified as the source of a \texttt{copy} 
instruction, normal memory is specified at the target of a \texttt{paste} instruction, or an attempt is made 
to access an accelerator that is not properly con-
figured for the software’s use; HR=0 only.

The access violates storage protection. In addition 
to the legacy VPM cases, this includes mis-
matches in access authority in which the pro-
cess-scoped PTE permits the access but the 
partition-scoped PTE does not. It also includes 
lack of necessary authority for accesses to pro-
cess-scoped tables, for example lack of write 
authority to set a reference bit in the pro-
cess-scoped PTE. (In such a case, the “access” 
reported as failing would be the access to the pro-
cess-scoped table. The HDAR would provide the 
guest real / (abbreviated) virtual address of the 
table entry.)

A Data Address Watchpoint match occurs, HR=0 
only.

An attempt is made to execute a \texttt{Load Atomic} or 
\texttt{Store Atomic} instruction with an invalid function 
code, HR=0 only.

A Hypervisor Data Storage interrupt also occurs when 
no higher priority exception exists and an attempt is 
made to execute a \texttt{Load Atomic} or \texttt{Store Atomic} 
instruction specifying an invalid function code.

**Programming Note**

When an attempt to execute a \texttt{Load Atomic} or 
\texttt{Store Atomic} instruction containing an invalid func-
tion code (see Figures 3 and 4 in Book II) causes 
a HDSI, the condition is very similar to an invalid 
form of an instruction. As a result, this instance of 
HDSI occurs with a high priority that blocks the 
translation process and prevents Reference and 
Change bit updates.

If a \texttt{stbcx}, \texttt{sthcx}, \texttt{stwcx}, \texttt{stdcx}, or \texttt{stqcx} 
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 
\texttt{Store} to the specified effective address would cause a 
Hypervisor Data Storage interrupt, it is implementa-
tion-dependent whether a Hypervisor Data Storage 
interrupt occurs.

If the XER specifies a length of zero for an indexed 
\texttt{Move Assist} instruction, a Hypervisor Data Storage 
interrupt does not occur.

The following registers are set:

<table>
<thead>
<tr>
<th>MSR</th>
<th>Set to 0.</th>
</tr>
</thead>
<tbody>
<tr>
<td>HSRR0</td>
<td>Set to the effective address of the instruction that caused the interrupt.</td>
</tr>
<tr>
<td>HSRR1</td>
<td>33:36 Set to 0.</td>
</tr>
<tr>
<td>Others</td>
<td>42:47 Set to 0.</td>
</tr>
<tr>
<td>MSR</td>
<td>Loaded from the MSR.</td>
</tr>
<tr>
<td>HDSISR</td>
<td>See Figure 65.</td>
</tr>
<tr>
<td>32</td>
<td>Set to 0.</td>
</tr>
</tbody>
</table>

---

**Chapter 6. Interrupts**

Page 1079
Set to 1 if the translation for an attempted access is not found in the Page Table; otherwise set to 0.

Set to 1 if the access is not permitted by Figure 44, 46, or the privilege, read, or read/write bits in Figure 45 as appropriate; otherwise set to 0.

Set to 1 if the access is due to a lwat, ldat, ldarx, lwat, lwarx, ldarx, lq, stq, lqarx, stwat, stdat, stdcx, stdat, stqcx, stbcx, stwcx, stbcx, stqcx, or stat 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.

Set to 1 by an explicit access for a Store, dbcz, 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.

Set to 0.

Set to 1 if a Data Address Watchpoint match occurs; otherwise set to 0.

Set to 1 if the access is not permitted by virtual page class key protection; otherwise set to 0.

Set to 0.

Set to 1 if an unsupported MMU configuration is found during the translation process.

Set to 1 if an attempt to atomically set a reference or change bit fails; otherwise set to 0.

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 5.7.12.

Set to 1 if HR=1 and the virtual / guest real address of a page directory entry, page table entry, or process table entry could not be translated; or HR=0, VPM=1, and the virtual address of a process table entry or segment table entry group could not be translated; otherwise set to 0.

Set to 0.

Set to 1 if an accelerator is specified as the source of a copy instruction, normal memory is specified as the target of a paste 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.

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.

Set to 0.

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 quadword for which access was attempted in the page that caused the exception, for a quadword Load or Store instruction (i.e., a Load or Store instruction for which the storage operand is a quadword; “first” refers to address order; see Section 6.7)
- a byte in the first aligned double-word for which access was attempted in the page that caused the exception, for a non-quadruple
Load or Store instruction
- set as described in the previous major bullet, except that the low order 5 bits are undefined, 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.

Programming Note

Note that for HPT translation, the full EA is a superset of the bits required to construct the full VA, when also provided with the VSID in the ASDR.

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. For a large segment the values of the bits below the VSID are undefined. 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 \( HDSISR_{10}=1 \)
- 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 when a process or segment table entry virtual address cannot be translated (the rest of the segment descriptor is implied).
- 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 in the partition-scoped table (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 LPCR_KBV=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 LPCR_KBV=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 66.
6.5.17 Hypervisor Instruction Storage Interrupt

A Hypervisor Instruction Storage interrupt occurs when 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, no higher priority exception exists, and either

(a) HPT translation is being performed, the value of the expression
\[ (\neg \text{MSR}_\text{IR}) \land (\text{VPM} \land \text{PRTE}_V \land \text{MSR}_\text{IR}) \]
is 1, and the next instruction to be executed cannot be fetched for any of the following reasons, or

(b) Radix Tree translation is being performed and partition-scoped translation prevents the next instruction to be executed from being fetched for any of the following reasons.

(In the expression for (a) above, “PRTE\(_V\)” is shorthand indicating that an invalid segment table descriptor did not stop the translation process. Note that an SLB hit may satisfy this condition even when the Process Table Entry is invalid.)

A Hypervisor Instruction Storage interrupt also occurs when no higher priority exception exists, HR=0, and a reference or change bit update cannot be performed as described below.

- Instruction address translation is enabled (MSR\(_\text{IR}\)=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 invalid radix tree configuration found in the partition-scoped tables, this type of exception will also be reported outside of hypervisor real mode for translation mode mismatches including UPRT=0 when HR=1, LPID=0 if MSR\(_\text{HV}\)=0 when HR=1, and HR=0 for LPID=0 when HR=1 for another partition ID.
- 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 (MSR\(_\text{IR}\)=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, this includes mismatches in access authority in which the process-scoped PTE permits the access but the partition-scoped PTE does not. It also includes lack of necessary authority for accesses to process-scoped tables, for example lack of write authority to set a reference bit in the process-scoped PTE. (In such a case, the “access” reported as failing would be the access to the process-scoped table. The HDAR would provide the guest real / (abbreviated) virtual address of the table entry.)

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

- Set to 1 if the translation for an attempted access is not found in the Page Table; otherwise set to 0.
- Set to 0.
- 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; otherwise set to 0.
- Set to 1 if the access is not permitted by Figure 44 46, or the read or read/write bits in Figure 45 as appropriate; otherwise set to 0.
- Set to 1 if the access is not permitted by virtual page class key protection; otherwise set to 0.
- Set to 1 if an unsupported MMU configuration is found during the translation process.
- 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 5.7.12.
46 Set to 1 if HR=1 and the guest real address of a page directory entry, page table entry, or process table entry could not be translated; or HR=0, VPM=1, and the virtual address of a process table entry or segment table entry group could not be translated; otherwise set to 0.

47 Set to 1 if the operation that caused the exception was attempting to update storage; otherwise set to 0. This bit may be set as a modifier to bit 45 to indicate that a change bit must be set. It may also be set as a modifier to bits 36 and 42, to indicate that write authority was required to complete the operation.

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. For a large segment the values of the bits below the VSID are undefined. When HR=1 (nested translation is taking place), set to 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 when a process or segment table entry virtual address cannot be translated (the rest of the segment descriptor is implied).
- 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 in the partition-scoped table (the effective address for other cases of the invalid MMU configuration exception will be found in HSR0)
- the guest real address of the process-scoped PTE when an attempt is made to set a reference bit without 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 65.

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 66.

6.5.18 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 MSR\textsubscript{WV PR}=0b00 and LPCR\textsubscript{EVRRT}=1, execution is attempted of a hypervisor privileged instruction or of an mtspr or mfspr instruction that specifies an SPR that is hypervisor privileged for the operation.
- When MSR\textsubscript{PR}=1, execution is attempted of an mtspr or mfspr instruction that specifies an SPR with spr\textsubscript{0}=0 that is not provided by the implementation.
- When MSR\textsubscript{PR}=0, execution is attempted of an mtspr or mfspr instruction that specifies SPR 0, 4, 5, or 6.
- When MSR\textsubscript{PR}=0 and LPCR\textsubscript{EVRRT}=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
33:36  Set to 0.
42:44  Set to 0.
45  Set to 1 for an attempt, when MSR$_{HV}$PR = 0b00 and LPCR$_{EVIRT}$=1, to execute a hypervisor privileged instruction or an \texttt{mtspr} or \texttt{mfspr} instruction that specifies an SPR that is hypervisor privileged for the operation; otherwise set to 0.
46:47  Set to 0.
Others  Loaded from the MSR.

MSR  See Figure 65 on page 1064.

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 MSR$_{HV}$PR = 0b00 and LPCR$_{EVIRT}$=1, it is implementation dependent whether HSRR1$_{45}$ 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 66.
This Programming Note illustrates how Hypervisor Emulation Assistance interrupts should be handled by software, including in environments that support nested hypervisors.

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\textsubscript{HV}\textsubscript{PR}=0b10 and that can access hypervisor resources directly; nested hypervisors run with MSR\textsubscript{HV}\textsubscript{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 LPC\textsubscript{REVIRT}=1 if nested hypervisors are used. (A Hypervisor Emulation Assistance interrupt can set HSRR\textsubscript{145} to 1 only when LPC\textsubscript{REVIRT}=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.
  - HSRR\textsubscript{145}=0: Emulate the instruction, recover from the error, or terminate this hypervisor, as appropriate.
  - HSRR\textsubscript{145}=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 level N+1 hypervisor.
    - HSRR\textsubscript{145}=0: Pass control to the level N+1 hypervisor as if the instruction had caused a Hypervisor Emulation Assistance interrupt (with HSRR\textsubscript{145}=0) to that hypervisor.
    - HSRR\textsubscript{145}=1:
      - The program that caused the interrupt is the level N+1 hypervisor: Virtualize the instruction.
      - The program that caused the interrupt is not the level N+1 hypervisor: Pass control to the level N+1 hypervisor as if the instruction had caused a Hypervisor Emulation Assistance interrupt (with HSRR\textsubscript{145}=1) to that hypervisor.

  - The program most recently dispatched by the level N hypervisor is an operating system.
    - HSRR\textsubscript{145}=0: Emulate the instruction if appropriate (rather than pass control to the operating system to do the emulation); otherwise 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 <xref to Section 6.5.9>.
    - HSRR\textsubscript{145}=1: Either terminate the operating system or pass control to the operating system as if the instruction had caused a Privileged Instruction type Program interrupt as described in a Programming Note near the end of <xref to Section 6.5.9>.

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\leq2). (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.)
6.5.19 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.

\[(MSREE | \neg (MSRHV) | MSRPR)\]

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 65 on page 1064.
| HMER | See Section 6.2.9 on page 1051.

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.

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.

### 6.5.20 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.

\[(MSREE | \neg (MSRHV))\]

Directed Hypervisor Doorbell exceptions are generated when Directed Hypervisor Doorbell messages (see Chapter 10) 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 | 33:36 Set to 0. 42:47 Set to 0. Others Loaded from the MSR.
| MSR | See Figure 65 on page 1064.
| HMER | See Section 6.2.9 on page 1051.

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 6.5.9, "Program Interrupt" on page 1074.

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 6.5.9, "Program Interrupt" on page 1074.

Programming Note

In versions of the architecture that precede V. 3.0B, an attempt when MSR_{PR}=0 to execute an mtstrup or mfspr instruction specifying an SPR that was not implemented (with the exception of SPR 0 for mtstrup and SPRs 0, 4, 5, and 6 for mfspr) was treated as a no-op. These former no-op cases now cause a Hypervisor Emulation Assistance interrupt (with HSRR1_{45}=0) when LPCR_{ EVIRT}=1 to enable future functions to be emulated on older implementations. (An attempt when MSR_{PR}=0 to execute an mtstrup instruction specifying SPRs 4, 5, and 6 now causes a Hypervisor Emulation Assistance interrupt regardless of the value of LPCR_{ EVIRT}.) If there is no future function emulation to be performed, hypervisor software must choose a policy from the following.

- treat the instruction as an error
- emulate the legacy no-op behavior
- give control to the operating system

Programming Note

Because the value of MSREE is always 1 when the thread is in problem state, the simpler expression

\[(MSREE | \neg (MSRHV))\]

is equivalent to the expression given above.

Programming Note

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 6.5.9, "Program Interrupt" on page 1074.

Programming Note

Because the value of MSREE is always 1 when the thread is in problem state, the simpler expression

\[(MSREE | \neg (MSRHV))\]

is equivalent to the expression given above.

Programming Note

Because the value of MSREE is always 1 when the thread is in problem state, the simpler expression

\[(MSREE | \neg (MSRHV))\]

is equivalent to the expression given above.
MSR See Figure 65 on page 1064.
Execution resumes at effective address 0x0000_0000_0000_0E80, possibly offset as specified in Figure 66.

Programming Note
Because the value of MSR\textsubscript{EE} is always 1 when the thread is in problem state, the simpler expression

\[(MSR_{EE} \mid \neg (MSR_{HV}))\]

is equivalent to the expression given above.

6.5.21 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.

\[(MSR_{EE} \mid \neg (MSR_{HV}) \mid MSR_{PR}) \& HVICE\]

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 65 on page 1064.
Execution resumes at effective address 0x0000_0000_0000_0EA0, possibly offset as specified in Figure 66.

6.5.22 Performance Monitor Interrupt
A Performance Monitor interrupt occurs when no higher priority exception exists, a Performance Monitor exception exists, event-based branches are disabled (MMCR\textsubscript{0}EE=0), and MSR\textsubscript{EE}=1, and either HFSCR\textsubscript{PR}=1 or the thread is in hypervisor state.

If multiple Performance Monitor exceptions occur before the first causes a Performance Monitor interrupt, the interrupt reflects the most recent Performance Monitor exception and the preceding Performance Monitor exceptions are lost.

The following registers are set:

SRR0 Set to the effective address of the instruction that would have been attempted to be execute next if no interrupt conditions were present.

SRR1
33:36 and 42:47 Reserved.
Others Loaded from the MSR.

MSR See Figure 65 on page 1064.
Execution resumes at effective address 0x0000_0000_0000_0F00, possibly offset as specified in Figure 66.

6.5.23 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 MSR\textsubscript{VEC}=0.

The following registers are set:

SRR0 Set to the effective address of the instruction that caused the interrupt.

SRR1
33:36 Set to 0.
42:47 Set to 0.
Others Loaded from the MSR.

MSR See Figure 65 on page 1064.
Execution resumes at effective address 0x0000_0000_0000_0F20, possibly offset as specified in Figure 66.

6.5.24 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 MSR\textsubscript{VSX}=0.

The following registers are set:

SRR0 Set to the effective address of the instruction that caused the interrupt.

SRR1
33:36 Set to 0.
42:47 Set to 0.
Others Loaded from the MSR.

MSR See Figure 65 on page 1064.
Execution resumes at effective address 0x0000_0000_0000_0F40, possibly offset as specified in Figure 66.
6.5.25 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 clrbhrb or mbhrbe instruction is executed in problem state when it has been made unavailable by MMCR0.
- the Transactional Memory Facility is accessed in any privilege state when it has been made unavailable by MSRTM.

The following registers are set:

- **SRR0**: Set to the effective address of the instruction that caused the interrupt.
- **SRR1**: 33:36 Set to 0. 42:47 Set to 0. Others Loaded from the MSR.
- **MSR**: See Figure 65 on page 1064.
- **FSCR**: 0:7 See Section 6.2.11 on page 1051. Others Not changed.

Execution resumes at effective address 0x0000_0000_0000_0F60, possibly offset as specified in Figure 66.

---

**Programming Note**

For the case of an outer `begin`, the interrupt handler should either return to the `begin` with MSR\textsubscript{TM} = 1 (allowing the program to use transactions), or treat the attempt to initiate an outer transaction as a program error.

---

6.5.26 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 stop instruction is executed in privileged non-hypervisor state when any of the following conditions exist.
  - \( \text{PSSCR}_{\text{EC}} = 1 \)
  - \( \text{PSSCR}_{\text{ESL}} = 1 \)
  - \( \text{PSSCR}_{\text{MTL}} \geq \text{PSSCR}_{\text{PSSL}} \)
  - \( \text{PSSCR}_{\text{RL}} \geq \text{PSSCR}_{\text{PSSL}} \)

The following registers are set:

- **HSRR0**: Set to the effective address of the instruction that caused the interrupt.
- **HSRR1**: 33:36 Set to 0. 42:47 Set to 0. Others Loaded from the MSR.
- **MSR**: See Figure 65 on page 1064.
- **HFSCR**: 0:7 See Section 6.2.12 on page 1052. Others Not changed.

Execution resumes at effective address 0x0000_0000_0000_0F80, possibly offset as specified in Figure 66.

---

6.5.27 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 65 on page 1064.

Execution resumes at the effective address specified in Figure 66.
When the System Call Vectored interrupt results in MSRIR being 1 or MSRHV being 0, the effective address described above is translated to a real address before being used to access storage. If the effective address cannot be translated, or if instructions cannot be fetched from the addressed storage location (e.g., the access would violate storage protection, or would be to No-execute storage), an [Hypervisor] Instruction Storage interrupt occurs before the first instruction at the effective address is executed.

Because the System Call Vectored interrupt uses save/restore registers that differ from those used by other interrupts, the System Call Vectored interrupt handler can run with address translation enabled and External interrupts enabled. Similarly, the Programming Note about managing MSRRI at the end of Section 6.4.3 does not apply to the System Call Vectored interrupt handler (the System Call Vectored interrupt does not alter MSRRI).
6.6 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 5.8.1, “Guarded Storage” and in Section 2.2 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 5.7.12.
- For a stbcx., sthcx., stwcx., stdcx., or 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 Load or 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 Load or 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 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 lq instruction, if RT+1 = RA then the contents of register RT+1 are not altered.
- For an update form Load or Store instruction, the contents of register RA are not altered.
6.7 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, if 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 and transaction failure due to attempted access of a disallowed type while in Transactional state 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.)

6.7.1 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 a storage protection exception.

1. System Reset
2. Machine Check except for those caused by an invalid attempt to access an accelerator

6.7.2 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 LPCR_{KBV}=1 and Virtualized Partition Memory is disabled by VPM=0 cause only a Hypervisor Data Storage exception (and never a Data Storage exception).

System-Caused or Imprecise

1. Program
   - Imprecise Mode Floating-Point Enabled Exception
2. Hypervisor Maintenance
**Instruction-Caused and Precise**

1. Instruction Segment
2. [Hypervisor] Instruction Storage or Machine Check for invalid accelerator access
3. Hypervisor Emulation Assistance or Program (Privileged Instruction)
4. Function-Dependent
   4.a Fixed-Point and Branch
      1 Hypervisor Facility Unavailable
      2 Facility Unavailable
      3a Program
         - Trap
         - TM Bad Thing
      3b System Call or System Call Vectored
3c.1 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
3c.2 all other Data Storage, Hypervisor Data Storage, [Hypervisor] Data Segment, Machine Check for invalid accelerator access, or Alignment
4 Trace
4.b Floating-Point
   1 Hypervisor Facility Unavailable
   2 Floating Point Unavailable
   3a Program
      - Precise Mode Floating-Pt Enabled Excep’n
   3b [Hypervisor] Data Storage, [Hypervisor] Data Segment, Machine Check for invalid accelerator access, or Alignment
4 Trace
4.c Vector
   1 Hypervisor Facility Unavailable
   2 Vector Unavailable
   3a [Hypervisor] Data Storage, [Hypervisor] Data Segment, Machine Check for invalid accelerator access, or Alignment
4 Trace
4.d VSX
   1 Hypervisor Facility Unavailable
   2 VSX Unavailable
   3a Program
      - Precise Mode Floating-Pt Enabled Excep’n
   3b [Hypervisor] Data Storage, [Hypervisor] Data Segment, Machine Check for invalid accelerator access, or Alignment
4 Trace
4.e Other Instructions
   1 Hypervisor Facility Unavailable
   2 Facility Unavailable
   3a [Hypervisor] Data Storage, [Hypervisor] Data Segment, Machine Check for invalid accelerator access, or Alignment
4 Trace
4 Trace

**Segment, Machine Check for invalid accelerator access, or Alignment**

**4 Trace**

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.

---

**6.8 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 7.2.1 of Book II.

---

**6.9 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 LPCRKBV=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 a storage protection exception.

   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 excep-

   tions 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). For instructions that are disallowed in Transactional state, and for mspr specifying an SPR that is not part of the checkpointed registers and is not the GSR or a Transactional Memory SPR, transaction failure takes priority over all interrupts except Privileged Instruction type Program interrupts, Hypervisor Emulation Assistance interrupts, and [Hypervisor] Facility Unavailable interrupts. For data accesses that are disallowed in Transactional state, transaction failure has the same priority as the group of "other" [Hypervisor] Data Storage, Data Segment, and Alignment exceptions. (See Section 5.3.1 of Book II.)

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
   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
   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
   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

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. TM instruction, \texttt{mt/fspr} specifying TM SPR
a. Program - Privileged Instruction (only for \texttt{treclaim} and \texttt{trechkpt})
b. Hypervisor Facility Unavailable
c. Facility Unavailable
d. Program - TM Bad Thing (only for \texttt{treclaim}, \texttt{trechkpt}, and \texttt{mtspr})
e. Trace

I. \texttt{rfid}, \texttt{hrfid}, \texttt{rfebb}, \texttt{rfscv}, and \texttt{mtmsr[d]}
a. These exceptions are mutually exclusive and have the same priority:
  - Program - Privileged Instruction, for all except \texttt{rfebb}
  - Hypervisor Emulation Assistance, for \texttt{hrfid} only
b. Hypervisor Facility Unavailable (\texttt{rfebb} only)
c. Facility Unavailable (\texttt{rfebb} only)
d. Program - TM Bad Thing for all except \texttt{mtmsr}
e. Program - Floating-Point Enabled Exception or all except \texttt{rfebb}
f. Trace, for \texttt{mtmsr[d]} and \texttt{rfebb} only

J. 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

K. Hypervisor Instruction Storage and Instruction Segment
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 two 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. Hypervisor Maintenance
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.

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 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=1 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.

**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.

### 6.10 Relationship of Event-Based Branches to Interrupts

#### 6.10.1 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.

#### 6.10.2 EBB Synchronization

When an event-based branch occurs, EBBRR 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 EBBRR has not completed execution.

### 6.10.3 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

7.
Chapter 7. Timer Facilities

7.1 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.

7.2 Time Base (TB)

The Time Base (TB) is a 64-bit register (see Figure 67) containing a 64-bit unsigned integer that is incremented periodically.

```
0 39
TBU40 ///
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 67. 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 67. When a mspsr 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 6 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_{PR}=1). If the means is under software control, it must be accessible only in hypervisor state (MSR_{HV_{PR}} = 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.
7.2.1 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 6 of Book II.

It is not possible to write the entire 64-bit Time Base using a single instruction. The 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 mtsp instruction; Figure 18.

The Time Base can be written by a sequence such as:

\[
\begin{align*}
\text{lwz } & \text{ Rx,upper # load 64-bit value for} \\
\text{lwz } & \text{ Ry,lower # TB into Rx and Ry} \\
\text{li } & \text{ Rz,0} \\
\text{mttbl } & \text{ Rx # set TBL to 0} \\
\text{mttbu } & \text{ Rx # set TBU} \\
\text{mttbl } & \text{ Ry # set TBL}
\end{align*}
\]

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.

\[
\begin{align*}
\text{lwz } & \text{ Rx,upper # load 64-bit Time Base value} \\
\text{clrldi } & \text{ Ry,Ry,40 # lower 24 bits of old TB} \\
\text{mttbu40 } & \text{ Rx # write upper 40 bits of TB} \\
\text{mftb } & \text{ Rz # read TB value again} \\
\text{clrldi } & \text{ Rx,Rz,40 # lower 24 bits of new TB} \\
\text{cmpld } & \text{ Rz,Ry # compare new and old lwr 24} \\
\text{bge } & \text{ done # no carry out of low 24 bits} \\
\text{addis } & \text{ Rx,Rx,0x0100} \\
\text{mttbu40 } & \text{ Rx # update to adjust for carry}
\end{align*}
\]

Providing 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.

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 6 of Book II for ways to compute time of day in POSIX format from the Time Base.

7.3 Virtual Time Base

The Virtual Time Base (VTB) is a 64-bit incrementing counter.

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

**Figure 68. Virtual Time Base**

Virtual Time Base increments at the same rate as the Time Base 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 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.
7.4 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.

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 `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 `mtspr` instruction.

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 63-d are ignored by the hardware.

**Programming Note**

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.

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.
7.4.1 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 18), 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.

7.5 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.

```
Programming Note

The maximum positive value supported by the Hypervisor Decrementer is 2^{h-1} - 1, represented with bits 0:64-h containing 0's and bits 65-h:63 containing 1's. The minimum value supported by the Hypervisor Decrementer is -2^{h-1}, represented as 0xFFFFFFFF_FFFF_FFFF.
```

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 0xFFFFFFFF_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.
```

7.6 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.

```
Figure 70. Processor Utilization of Resources Register

The PURR is a hypervisor resource; see Chapter 2.
```
The contents of the PU RR 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.

#### 7.7 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.

![Figure 71. Scaled Processor Utilization of Resources Register](image)

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 \( S_{ab}/(T_{ab} \times f_r \times (1 - c_r)) \) 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.

---

### 7.8 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).

![Instruction Counter](image)

*Figure 72. Instruction Counter*
Chapter 8. Debug Facilities

8.1 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 8.2), Completed Instruction Address Breakpoint Register (see Section 8.3), and the Data Address Watchpoint Register (DAWRn) and Data Address Watchpoint Register Extension (DAWRXn) (see Section 8.4). The interrupt associated with the Data Address Breakpoint registers is described in Section 6.5.3. The interrupt associated with the Completed Instruction Address Breakpoint Register is described in Section 6.5.15. The Trace facility, which can be used for single-stepping as well as for control flow tracing, is described in Section 6.5.15.

The mfspr and mtsp r instructions (see Section 4.4.4) 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).

8.2 Come-From Address Register

The Come-From Address Register (CFAR) is a 64-bit register. When an rfebb, rfid, or rsccv instruction is executed, the register is set to the effective address of the instruction. When a 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 Branch instruction, except that if the Branch instruction is a B-form Branch (i.e., bc, bca, bcl, or bcla) for which the target address is in the instruction cache block containing the Branch instruction or is in the previous or next cache block, the register is not necessarily set. For Branch instructions, the setting need not occur until a subsequent context synchronizing operation has occurred.

<table>
<thead>
<tr>
<th>CFAR</th>
<th>//</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>62 63</td>
</tr>
</tbody>
</table>

Figure 73. Come-From Address Register

The contents of the CFAR can be read and written using the mfspr and mtsp r 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.

8.3 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 75.

<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></td>
</tr>
</tbody>
</table>

Bit(s) Name Description
0:61 CIEA Completed Instruction Effective Address
62:63 PRIV Privilege
  00: Disable matching
  01: Match in problem state
  10: Match in privileged (non-hypervisor) state
  11: Match in hypervisor state

A Completed Instruction Address Breakpoint match occurs upon instruction completion if all of the following conditions are satisfied:
- the completed instruction address is equal to CIEA_{0:61} || 0b00.
- the thread run level matches that specified in RLM.

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 provided that no higher priority interrupt occurs from the completion of the instruction (see Section 6.5.15).

### 8.4 Data Address Watchpoint

The Data Address Watchpoint mechanism provides a means of detecting load and store accesses to a range of addresses starting at a designated doubleword. The address comparison is done on an effective address (EA).

#### 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.

The Data Address Watchpoint mechanism is controlled by a single set of SPRs, numbered with n=0: the Data Address Watchpoint Register (DAWRn), shown in Figure 75, and the Data Address Watchpoint Register Extension (DAWRXn), shown in Figure 76.

<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:60</td>
<td>DEAW</td>
<td>Data Effective Address Watchpoint</td>
</tr>
</tbody>
</table>

#### Figure 75. Data Address Watchpoint Register

<table>
<thead>
<tr>
<th>Bit(s) Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>48:53 MRD</td>
<td>Match Range in Doublewords biased by -1. (0b0000000 = 1 DW, 0b1111111 = 64 DW)</td>
</tr>
<tr>
<td>56 HRAMMC</td>
<td>Hypervisor Real Addressing Mode Match Control</td>
</tr>
<tr>
<td>57 DW</td>
<td>Data Write</td>
</tr>
<tr>
<td>58 DR</td>
<td>Data Read</td>
</tr>
<tr>
<td>59 WT</td>
<td>Watchpoint Translation</td>
</tr>
<tr>
<td>60 WTI</td>
<td>Watchpoint Translation Ignore</td>
</tr>
<tr>
<td>61:63 PRIVM</td>
<td>Privilege Mask</td>
</tr>
<tr>
<td>61 HYP</td>
<td>Hypervisor state</td>
</tr>
<tr>
<td>62 PNH</td>
<td>Privileged but Non-Hypervisor state</td>
</tr>
<tr>
<td>63 PRO</td>
<td>Problem state</td>
</tr>
</tbody>
</table>

All other fields are reserved.

#### Figure 76. Data Address Watchpoint Register Extension

The supported PRIVM values are 0b000, 0b001, 0b010, 0b011, 0b100, and 0b111. If 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.
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.

- the access is
  - a quadword access and located in the range
    \[(DEAW_{0:59} || 0b0) \leq (EA_{0:59} || 0b0) \leq (DEAW_{0:59} || 0b0) + (550 || MRD_{0:4} || 0b0)\]
    such that \((EA_{0:60} AND (551 || 60)) = (DEAW_{0:60} AND (551 || 60)).\)
  - not a quadword access and located in the range
    \[DEAW_{0:60} \leq EA_{0:60} \leq (DEAW_{0:60} + (550 || MRD_{0:5}))\]
    such that \((EA_{0:60} AND (551 || 60)) = (DEAW_{0:60} AND (551 || 60)).\)

- \((MSRDR = DAWRXnWT) | DAWRXnWTI\)

- the thread is in
  - hypervisor state and DAWRXnHYP = 1, or
  - privileged but non-hypervisor state and DAWRXnPNH = 1, or
  - problem state and DAWRXnPR = 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 32-bit mode the high-order 32 bits of the EA are treated as zeros for the purpose of detecting a match.

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 6.5.3, "Data Storage Interrupt" on page 1069 and Section 6.5.16, "Hypervisor Data Storage Interrupt" on page 1078). 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.

**Programming Note**

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 DAWRXn60:63 to 0b0111.

**Programming Note**

The Data Address Watchpoint mechanism does not apply to instruction fetches.
Chapter 9. Performance Monitor Facility

9.1 Overview

The Performance Monitor facility provides a means of collecting information about program and system performance.

9.2 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, and MMCRA (Monitor Mode Control Registers 0, 1, 2, and A), which control the Performance Monitor facility
  - SIAR, SDAR, and SIER (Sampled Instruction Address Register, Sampled Data Address Register, and Sampled Instruction Event Register), which contain the address of the “sampled instruction” and of the “sampled data,” and additional information about the “sampled instruction” (see Section 9.4.8 - Section 9.4.10).

- the Performance Monitor interrupt and Performance Monitor event-based branch, which can be caused by monitored conditions and events.

Many aspects of the operation of the Performance Monitor are summarized by the following hierarchy, which is described starting at the lowest level.

- 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 9.4.3). A single Performance Monitor alert may reflect multiple enabled conditions and events.

- When a Performance Monitor alert occurs, MMCR0PMAO is set to 1 and the writing of BHRB entries, if in process, is suspended.

When the contents of MMCR0PMAO change from 0 to 1, a Performance Monitor exception will come into existence within a reasonable period of time. When the contents of MMCR0PMAO 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, MMCRI_{EBE} = 0, and either HFSCR_{PM}=1 or the thread is in hypervisor state, an interrupt occurs.
  - If MSR_{PR} = 1, MMCRI_{EBE} = 1, a Performance Monitor event-based exception occurs if BESCR_{PM}=1, provided that event-based exceptions are enabled by FSCR_{EBB} and HFSCR_{EBB}. When a Performance Monitor...
event-based exception occurs, an event-based branch is generated if BES- 
CRGE=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 9.4.4.

### 9.3 No-op Instructions Reserved for the Performance Monitor

The following forms of the **and** \( x,x,x \) instruction are reserved for exclusive use by the Performance Monitor. \( \text{and} \ x,x,x; \) 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.

**MMCR1:**
- PMC1SEL=E0 (count PMC1 on dispatch)
- PMC4SEL=E0 (count PMC4 on completion)
- Other counters initialized as desired.

**MMCR2:** 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.

### 9.4 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 **mfspr** and may reflect some events due to instructions that follow the **mfspr**. 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 **mfspr** is executed. Similarly, if an **mtspr** instruction is executed that changes the contents of the Time Base, the change is not guaranteed to have taken effect with respect to causing Time Base transition events until after a subsequent context synchronizing instruction has been executed.

If an **mtspr** instruction is executed that changes the value of a Performance Monitor register other than SIAR, SDAR, and SIER, the change is not guaranteed to have taken effect until after a subsequent context synchronizing instruction has been executed (see Chapter 11. “Synchronization Requirements for Context Alterations” on page 1133).

**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.

### 9.4.1 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., SIER, SIAR, SDAR, and MMCR1 at SPR numbers 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 4.4.4 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 4.4.4. (All 32-bit PowerPC implementations used a different set.

### 9.4.2 Performance Monitor Counters

The six Performance Monitor Counters, PMC1 through PMC6, are 32-bit registers that count events.

<table>
<thead>
<tr>
<th>PMC1</th>
<th>PMC2</th>
<th>PMC3</th>
<th>PMC4</th>
<th>PMC5</th>
<th>PMC6</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>63</td>
</tr>
</tbody>
</table>

**Figure 77. Performance Monitor Counter registers**

PMC1 - PMC4 are referred to as “programmable” counters since the events that can be counted can be specified by the program. The events that are counted by each counter are specified in MMCR1.

PMC5 and PMC6 are not programmable and can be specified as being part of the Performance Monitor Facility or not part of it. PMC5 counts instructions completed, and PMC6 counts cycles. The PMCC field in MMCR0 controls whether or not PMCs 5-6 are part of the Performance Monitor Facility, and the result of accessing these counters when they are not part of the Performance Monitor Facility.

--- 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.

--- 9.4.2.1 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 MMCR1.

Randomly sampled events can cause the counters to count only when random sampling has been enabled by setting MMCR0SE=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 MMCR1.
9.4.3 Threshold Event Counter

The threshold event counter and associated controls are in MMCRA (see Section 9.4.7). When Performance Monitor alerts are enabled (MMCR0PMAE=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 MMCR0PMAE to 1.

Programming Note

A typical sequence of operations that enables use of the PMCs is as follows:
- Freeze the counters by setting MMCR0FC=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 MMCR1.
- 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 MMCR0FC to 0 to disable freezing the counters, and set MMCR0PMAE to 1 if a Performance Monitor alert (and the corresponding Performance Monitor interrupt) is desired when an enabled condition or event occurs. (See Section 9.2 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 9.4.4 - 9.4.10 for information regarding MMCRs, SIAR, SDAR, and SIER, and some additional usage examples.

Programming Note

Because hardware can modify the contents of the threshold event counter when random sampling is enabled (MMCRASE=1) and MMCR0PMAE=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 MMCRA_TECC, 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), within 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.

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.
9.4.4 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>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>63</td>
</tr>
</tbody>
</table>

Figure 78. 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 PMCC=0b10 or 0b11, problem state programs have write access to MMCR0 in order to enable event-based branch routines to reset the FC bit after it has been set to 1 as a result of an enabled condition or event (FCECE=1). During event processing, the event-based branch handler would write the desired initial values to the PMCs and reset the FC bit to 0. PMAO and PMAE can also be set to their appropriate values during the same write operation before returning.

The bit definitions of MMCR0 are as follows.

<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>Freeze Counters (FC)</td>
</tr>
<tr>
<td>0</td>
<td>The PMCs are incremented (if permitted by other MMCR bits).</td>
</tr>
<tr>
<td>1</td>
<td>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>
</tr>
<tr>
<td>33</td>
<td>Freeze Counters and BHRB in Privileged State (FCS)</td>
</tr>
<tr>
<td>0</td>
<td>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 MMCRA).</td>
</tr>
<tr>
<td>34</td>
<td>Freeze Counters and BHRB in Problem State (FCP)</td>
</tr>
<tr>
<td>0</td>
<td>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 MMCRA).</td>
</tr>
<tr>
<td>1</td>
<td>The PMCs are not incremented, and entries are not written into the BHRB, if MSRPR=0b01.</td>
</tr>
<tr>
<td></td>
<td>If the value of bit 51 (FCPC) is 0, this field has the following meaning.</td>
</tr>
<tr>
<td></td>
<td>0 The PMCs are not incremented, and entries are not written into the BHRB, if MSRPR=0b00.</td>
</tr>
<tr>
<td></td>
<td>Conditionally Freeze Counters and BHRB in Problem State (FCP)</td>
</tr>
<tr>
<td></td>
<td>If the value of bit 51 (FCPC) is 1, this field has the following meaning.</td>
</tr>
<tr>
<td></td>
<td>0 The PMCs are not incremented, and entries are not written into the BHRB, if MSRPR=0b00.</td>
</tr>
<tr>
<td></td>
<td>1 The PMCs are not incremented, and entries are not written into the BHRB, if MSRPR=0b01.</td>
</tr>
<tr>
<td></td>
<td>If the value of bit 51 (FCPC) is 1, this field has the following meaning.</td>
</tr>
<tr>
<td></td>
<td>0 The PMCs are not incremented, and entries are not written into the BHRB, if MSRPR=0b00.</td>
</tr>
<tr>
<td></td>
<td>1 The PMCs are not incremented, and entries are not written into the BHRB, if MSRPR=0b01.</td>
</tr>
<tr>
<td>35</td>
<td>Free Counters while Mark = 1 (FCM1)</td>
</tr>
<tr>
<td>0</td>
<td>The PMCs are incremented (if permitted by other MMCR bits).</td>
</tr>
<tr>
<td>1</td>
<td>The PMCs are not incremented if MSRPM=1.</td>
</tr>
<tr>
<td>36</td>
<td>Free Counters while Mark = 0 (FCM0)</td>
</tr>
<tr>
<td>0</td>
<td>The PMCs are incremented (if permitted by other MMCR bits).</td>
</tr>
<tr>
<td>1</td>
<td>The PMCs are not incremented if MSRPM=0.</td>
</tr>
<tr>
<td>37</td>
<td>Performance Monitor Alert Enable (PMAE)</td>
</tr>
<tr>
<td>0</td>
<td>Performance Monitor alerts are disabled and BHRB entries are not written.</td>
</tr>
<tr>
<td>1</td>
<td>Performance Monitor alerts are enabled, and BHRB entries are written (if enabled by other bits) until a Performance Monitor alert occurs, at which time:</td>
</tr>
<tr>
<td></td>
<td>▪ MMCR0PMAE is set to 0</td>
</tr>
<tr>
<td></td>
<td>▪ MMCR0PMAO is set to 1</td>
</tr>
</tbody>
</table>
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 $\text{MMCR0TRIGGER} = 0$, at which time:
  - If $\text{MMCR0TRIGGER} = 0$, $\text{FCECE}$ is set to 1

If the enabled condition or event occurs when $\text{MMCR0TRIGGER} = 1$, 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.

Programming Note

Software can set this bit and $\text{MMCR0PMAO} = 0$ to prevent Performance Monitor exceptions.

Software can set this bit to 1 and then poll the bit to determine whether an enabled condition or event has occurred. This is especially useful for software that runs with $\text{MSR}_{EE} = 0$.

In earlier versions of the architecture that lacked the concept of Performance Monitor alerts, this bit was called Performance Monitor Exception Enable (PMXE).

Time Base Event Enable (TBEE)

- **0** Time Base transition events are disabled.
- **1** Time Base transition events are enabled.

Programming Note

When $\text{PMC3}$ is configured to count the occurrence of Time Base transition events, the events are counted regardless of the value of $\text{MMCR0TBEE}$. (See Section 9.4.5.) The occurrence of a Time Base transition causes a Performance Monitor alert only if $\text{MMCR0TBEE} = 1$.

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** $\text{clrbhrb}$ and $\text{mfhrb}$ are not available in problem state.
- **1** $\text{clrbhrb}$ and $\text{mfhrb}$ 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 $\text{BESCR}_{PME}$.
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 MMCRR0PMAE (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 9.4.1) of non-privileged SPRs in problem state as described below.

**Programming Note**

The PMCC field does not affect the behavior of the privileged Performance Monitor registers (SPRs 784-792, 795-798); accesses to these SPRs in problem state result in Privileged Instruction type Program interrupts.

The PMCC field also does not affect the behavior of write operations to group B; write operations to SPRs in group B are treated as not supported regardless of privilege state. See the `mtspr` instruction description in Section 4.4.4 for additional information on accessing SPRs that are not supported.

00  PMCs 5 - 6 are included in the Performance Monitor.
Groups A and B are read-only 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 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 9.4.2 for details.

Group A except for PMCs 5-6 (SPRs 775,776) 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, in problem state, to read or write to PMCs 5-6 (SPRs 775,776), or to read from MMCR1, a Facility Unavailable interrupt will occur.

When an SPR is made available by the PMCC field, it is available only if it has not been made unavailable by the HFSCR (see Section 6.2.12).
Freeze Counters in Transactional State (FCTS)

0 PMCs are incremented (if permitted by other MMCR bits).
1 PMCs are not incremented when the thread is in Transactional state.

Freeze Counters in Non-Transactional State (FCNTS)

0 PMCs are incremented (if permitted by other MMCR bits).
1 PMCs are not incremented when the thread is in Non-transactional state.

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.

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

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 6.2.11 for additional information.

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.

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.

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 6.2.11 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.

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.

50

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 PMCs 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 PMCs 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 PMCs when PMC1 becomes negative, and cause a Performance Monitor interrupt without freezing any PMCs. The PMCs 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 MSRHV, MMCR0[FCPC] must be set to 0.

**Performance Monitor Alert Qualifier** (PMAQ)

This bit provides additional implementation-dependent information about the cause of the Performance Monitor alert. When a Performance Monitor alert occurs, this bit is set to 0 if no additional information is available.

**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.

**Freeze Counters in Suspended State** (FCSS)

0: PMCs are incremented (if permitted by other MMCR bits).
1: PMCs are not incremented when the thread is in Suspended state.

**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.
Freeze Counters and BHRB in Hypervisor State (FCH)

0  The PMCs are incremented (if permitted by other MMCR bits) and BHRB entries are written (if permitted by the BHRB Instruction Filtering Mode field in MMCRA).
1  The PMCs are not incremented and BHRB entries are not written if MSR_WV.PR=0b10.

9.4.5 Monitor Mode Control Register 1

Monitor Mode Control Register 1 (MMCR1) is a 64-bit register as shown below.

<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 (MMCRASE=1). The sampling modes corresponding to each event are listed in parentheses. (The sampling mode is specified in MMCRASM.)

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 5). (RIS, RLS, RBS)
E8  The threshold event counter has exceeded the number of events corresponding to threshold E (see Table 5). (RIS, RLS, RBS)
EA  The thread filled a block in a data cache with data that were accessed by a randomly sampled Load instruction. (RIS, RLS)
EC  The threshold event counter has reached its maximum value. (RIS, RLS, RBS)

The following events can occur regardless of whether random sampling is enabled.

F0  A cycle has occurred. This event is not controlled by MMCR0.FC1-4WAIT.
F2  A cycle has occurred in which the thread completed one or more instructions.
F4  The thread has completed a Floating-Point, Vector Floating-Point, or VSX Floating-Point instruction other than a

### Table 5: 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>

Programming Note

When PMC 1 is counting cycles, it is not controlled by this bit. See the description of the F0 event in Section 9.4.5.
Load or Store instruction to the point at which it has reported all exceptions it will cause.

F6 The thread has failed to locate an ERAT entry during instruction address translation.

F8 A cycle has occurred during which all previously initiated instructions have completed and no instructions are available for initiation.

FA A cycle has occurred during which the RUN bit of the CTRL register for one or more threads of the multi-threaded processor was set to 1.

FC A load type instruction finished. If the instruction caused more than one reference, only one will be counted.

FE The thread has completed an instruction.

**40:47 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-D</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

The following events can occur only when random sampling is enabled (MMCRASE=1). The sampling modes corresponding to each event are listed in parentheses. (The sampling mode is specified in MMCRASM.)

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 5). (RIS, RLS, RBS)

E6 The threshold event counter has exceeded the number of events corresponding to threshold F (see Table 5). (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 completed a Branch instruction for which the branch was taken.

FC The thread has failed to locate an instruction in the primary 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 cache.

**48:55 PMC3 Selector (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-D</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

The following events can occur only when random sampling is enabled (MMCRASE=1). The sampling modes corresponding to each event are listed in parentheses. (The sampling mode is specified in MMCRASM.)

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)

E4 The thread has mispredicted either whether or not the branch would be taken, or if taken, the target address of a randomly sampled Branch instruction. (RIS, RBS)

E6 The thread has failed to locate an ERAT entry during data address translation for a randomly sampled instruction. (RIS, RLS)

E8 The threshold event counter has exceeded the number of events corresponding to threshold C (see Table 5). (RIS, RLS, RBS)

EA The threshold event counter has exceeded the number of events corresponding to threshold G (see Table 5). (RIS, RLS, RBS)
The following events can occur regardless of whether random sampling is enabled.

F0  The thread has attempted to store data in the primary data cache but no block corresponding to the real address existed.
F2  The thread has dispatched an instruction.
F4  The thread has completed an instruction when the RUN bit of the CTRL register for all threads on the multi-threaded processor contained 1.
F6  The thread has filled a block in the primary data cache with data that were accessed by a Load instruction.
F8  A Time Base transition event has occurred for the thread. This event is counted regardless of whether or not Time Base transition events are enabled by MMCR0.TBE.
FA The thread has loaded an instruction from a higher level cache than the tertiary cache.
FC The thread was unable to translate a data virtual address using the TLB.
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.

56:63  **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-CF</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

The following events can occur only when random sampling is enabled (MMCRAE=1). The sampling modes corresponding to each event are listed in parentheses. (The sampling mode is specified in MMCRA.SM.)

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 5). (RIS, RLS, RBS)
EC  The threshold event counter has exceeded the number of events corresponding to threshold H (see Table 5). (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 MMCR0 and MMCR1. PMC1-8 were all programmable.

If more programmable PMCs are implemented in the future, additional MCMRs may be defined to cover the additional selectors.

#### 9.4.6 Monitor Mode Control Register 2

Monitor Mode Control Register 2 (MMCR2) 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 80. Monitor Mode Control Register 2*
When MMCR0PMCC = 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 MMCR0PMCC is set to 0b10 or 0b11, providing problem state programs read/write access to MMCR2, only the FCnP0 bits can be accessed. All other bits are not changed when mspr 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 (FCnS)</td>
</tr>
<tr>
<td></td>
<td>0 PMCnt is incremented (if permitted by other MMCR bits).</td>
</tr>
<tr>
<td></td>
<td>1 PMCnt is not incremented if MSRnHV PR=0b00.</td>
</tr>
<tr>
<td>1</td>
<td>Freeze Counter n in Problem State if MSRnHV=0 (FCnP0)</td>
</tr>
<tr>
<td></td>
<td>0 PMCnt is incremented (if permitted by other MMCR bits).</td>
</tr>
<tr>
<td></td>
<td>1 PMCnt is not incremented if MSRnHV PR=0b01.</td>
</tr>
<tr>
<td>2</td>
<td>Freeze Counter n in Problem State if MSRnHV=1 (FCnP1)</td>
</tr>
<tr>
<td></td>
<td>0 PMCnt is incremented (if permitted by other MMCR bits).</td>
</tr>
<tr>
<td></td>
<td>1 PMCnt is not incremented if MSRnHV PR=0b11.</td>
</tr>
<tr>
<td>3</td>
<td>Freeze Counter n while Mark = 1 (FCnM1)</td>
</tr>
<tr>
<td></td>
<td>0 PMCnt is incremented (if permitted by other MMCR bits).</td>
</tr>
<tr>
<td></td>
<td>1 PMCnt is not incremented if MSRnPMM=1.</td>
</tr>
<tr>
<td>4</td>
<td>Freeze Counter n while Mark = 0 (FCnM0)</td>
</tr>
<tr>
<td></td>
<td>0 PMCnt is incremented (if permitted by other MMCR bits).</td>
</tr>
<tr>
<td></td>
<td>1 PMCnt is not incremented if MSRnPMM=0.</td>
</tr>
<tr>
<td>5</td>
<td>Freeze Counter n in Wait State (FCnWAIT)</td>
</tr>
<tr>
<td></td>
<td>0 PMCnt is incremented (if permitted by other MMCR bits).</td>
</tr>
<tr>
<td></td>
<td>1 PMCnt is not incremented if CTRLnRUN=0.</td>
</tr>
</tbody>
</table>

### Programming Note

The operating system is expected to set CTRLnRUN to 0 when the thread is in a “wait state”, i.e., when there is no process ready to run.

6 Freeze Counter n in Hypervisor State (FCnH)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Freeze Counter n in Hypervisor State (FCnH)</td>
</tr>
<tr>
<td></td>
<td>0 PMCnt is incremented (if permitted by other MMCR bits).</td>
</tr>
<tr>
<td></td>
<td>1 PMCnt is not incremented if MSRnHV PR=0b10.</td>
</tr>
</tbody>
</table>

Bits 54:63 of MMCR2 are reserved.

### 9.4.7 Monitor Mode Control Register A

Monitor Mode Control Register A (MMCRA) is a 64-bit register as shown below.

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

#### Figure 81. Monitor Mode Control Register A

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 mspr is executed in problem state.

### 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 mspr because future versions of the architecture may allow various fields of MMCRA to be modified in problem state.

<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:31</td>
<td>Problem state access (SPR 770)</td>
</tr>
<tr>
<td></td>
<td>Reserved</td>
</tr>
<tr>
<td>32:33</td>
<td>BHRB Instruction Filtering Mode (IFM)</td>
</tr>
</tbody>
</table>

The bit definitions of MMCRA are as follows.

#### Programming Note

The operating system is expected to set CTRLnRUN to 0 when the thread is in a “wait state”, i.e., when there is no process ready to run.
This field controls the filter criterion used by the hardware when recording Branch instructions into the BHRB. See Section 9.5.

00 All taken Branch instructions are entered into the BHRB unless prevented by other filtering fields.

01 Do not record any Branch instructions in which the LK field is set to 0.

10 Do not record I-Form instructions. For B-Form and XL-Form instructions for which the BO field indicates “Branch always,” do not record the instruction if it is B-Form and do not record the instruction address but record only the branch target address if it is XL-Form.

11 Filter and enter BHRB entries as for mode 10, but for B-Form and XL-Form instructions for which BO<0>=1 or for which the “a” bit in the BO field is set to 1, do not record the instruction if it is B-Form and do not record the instruction address but record only the branch target address if it is XL-Form.

---

### 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.

---

#### Threshold Event Counter Exponent (TECX)

This field species the exponent of the threshold event counter value. See Section 9.4.3 for additional information. The maximum exponent supported is at least 5.

34:36

#### Threshold Event Counter Multiplier (TECM)

This field species the multiplier of the threshold event counter value. See Section 9.4.3 for additional information.

37:38

---

#### 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.

45:47

---

#### 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<31>=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<30>.)

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)
The thread has failed to locate data for a randomly sampled Load instruction in the primary data cache. (RIS, RLS)

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 MMCRASE=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_{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

Eligibility for Random Sampling (ES)
When random sampling is enabled (MMCRASE=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

When random sampling is enabled (MMCRASE=1) and the SM field indicates random Load/Store Facility sampling (RLS), the encodings of this field specify the instructions that are eligible to be sampled as follows.

000 Instructions for which the thread has attempted to load data from the data cache but no block corresponding to the real address existed.
001 Reserved
010 Reserved
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

When random sampling is enabled (MMCRASE=1) and the SM field indicates random Branch Facility sampling (RBS), the encodings of this field specify the instructions that are eligible to be sampled as follows.

000 Instructions for which the thread has either mispredicted whether or not the branch would be taken, or if taken, the target address of a Branch instruction.
001 Instructions for which the thread has mispredicted whether or not the branch of a Branch instruction would be taken because the contents of the Condition Register differed from the predicted contents.
010 Instructions for which the thread has mispredicted the target address of a Branch instruction.
011 All Branch instructions for which the branch was taken.

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

63 Random Sampling Enable (SE)
0 Random sampling is disabled.
1 Random sampling is enabled.

See Section 9.4.2.1 for information about random sampling.

9.4.8 Sampled Instruction Address Register

The Sampled Instruction Address Register (SIAR) is a 64-bit register.

When a Performance Monitor alert occurs because of an event caused by execution of a randomly sampled instruction, the SIAR contains the effective address of the instruction if SIERSIARV = 1 and contains an undefined value if 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”.

The contents of SIAR may be altered by the hardware if and only if MMCR0PMAE=1. Thus after the Performance Monitor alert occurs, the contents of SIAR are not altered by the hardware until software sets MMCR0PMAE to 1. After software sets MMCR0PMAE to 1, the contents of SIAR are undefined until the next Performance Monitor alert occurs.

Programming Note
When the Performance Monitor alert occurs, SIERAMPPRaminsamphv indicates the value of MSRHPR that was in effect when the sampled instruction was being executed. (The contents of these SIER bits are visible only in privileged state.)

9.4.9 Sampled Data Address Register

The Sampled Data Address Register (SDAR) is a 64-bit 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 SIERSDARV = 1 and contains an undefined value if 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.”

The contents of SDAR may be altered by the hardware if and only if MMCR0PMAE=1. Thus after the Performance Monitor alert occurs, the contents of SDAR are not altered by the hardware until software sets MMCR0PMAE to 1. After software sets MMCR0PMAE to 1, the contents of SDAR are undefined until the next Performance Monitor alert occurs.
9.4.10 Sampled Instruction Event Register

The Sampled Instruction Event Register (SIER) is a 64-bit register.

| SIER | 0 | 63 |

Figure 84. 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 contents of 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) | Implementation-dependent |

38 Sampled MSR\(_{PP}\) (SAMPPR)

Value of MSR\(_{PP}\) when the Performance Monitor alert occurred.

39 Sampled MSR\(_{HV}\) (SAMPHV)

Value of MSR\(_{HV}\) when the Performance Monitor alert occurred.

40 Reserved

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.

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.

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.

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.

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.

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 |

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. |
The thread obtained the instruction in the secondary cache.

The thread obtained the instruction in the tertiary cache.

The thread failed to obtain the instruction in the primary, secondary, or tertiary cache.

Reserved

Reserved

Reserved

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.

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.

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.

The instruction was not a mispredicted Branch instruction.

The thread mispredicted whether or not the branch would be taken, or if taken, the target address.

Reserved

Sampled Instruction Data ERAT Miss (SID-ERAT)

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.

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.

The instruction did not require data address translation.

The thread translated the data virtual address using the TLB.

A PTEG required for data address translation for the instruction was obtained from the secondary cache.

A PTEG required for data address translation for the instruction was obtained from the tertiary cache.

A PTEG required for data address translation for the instruction was obtained from storage that did not reside in any cache.

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.

A PTEG required for data address translation for the instruction was obtained from a cache on a different chip from the thread.

Reserved

Sampled Instruction Data Storage Access Information (SIDSAI)

This field contains information about data storage accesses made by the sampled instruction. The values and their meanings are as follows.

The instruction did not require data address translation.

The instruction was a Read for which the thread obtained the referenced data from the primary data cache.

The instruction was a Read for which the thread obtained the referenced data from the secondary cache.

The instruction was a Read for which the thread obtained the referenced data from storage that did not reside in any cache.

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.

The instruction was a Read for which the thread obtained the referenced data from a cache on a different chip from the thread.

The instruction was a Store for which the data were placed into a location other than the primary data cache.
Chapter 9. Performance Monitor Facility

### 9.5 Branch History Rolling Buffer

The Branch History Rolling Buffer (BHRB) is described in Chapter 8 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, MMCR0\textsubscript{BHRBA} must be set to 1 to enable execution of \texttt{clrhrb} and \texttt{mfbhrbe} instructions in problem state. Additionally, MMCR0\textsubscript{PMCC} must be set to 0b10 or 0b11 to allow problem state programs to read and write the necessary Performance Monitor registers. (See Section 9.4.4.)

If Performance Monitor event-based branching is desired, MMCR0\textsubscript{EBE} must also be set to 1 to enable Performance Monitor event-based branches.

#### Programming Note

Enabling Performance Monitor event-based branching eliminates the need for the problem state program to poll MMCR0\textsubscript{PMAE} in order to determine when a Performance Monitor alert occurs.

The BHRB is written by the hardware if and only if Performance Monitor alerts are enabled by setting MMCR0\textsubscript{PMAE} to 1. After MMCR0\textsubscript{PMAE} has been set to 1 and a Performance Monitor alert occurs, MMCR0\textsubscript{PMAE} is set to 0 and the BHRB is not altered by hardware until software sets MMCR0\textsubscript{PMAE} to 1 again.

When MMCR0\textsubscript{PMAE}=1, \texttt{mfbhrbe} instructions return 0s to the target register.

#### Programming Note

\texttt{mfbhrbe} instructions return 0s when MMCR0\textsubscript{PMAE}=1 in order to prevent software from reading the BHRB while it is being written by hardware.

#### BHRB Filtering

When the BHRB is written by hardware, only those Branch instructions that meet the filtering criterion specified in MMCRA\textsubscript{IFM} and for which the branch was taken are included.

### 9.6 Interaction With Other Facilities

If tracing is active (MSR\textsubscript{SE}=1 or MSR\textsubscript{BE}=1), the contents of SIAR and SDAR as used by the Performance Monitor facility are undefined and may change even when MMCR0\textsubscript{PMAE}=0.

#### Programming Note

A potential combined use of the Trace and Performance Monitor facilities is to trace the control flow of a program and simultaneously count events for that program.
Chapter 10. Processor Control

10.1 Overview

The Processor Control facility provides a mechanism for the 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.

10.2 Programming Model

Both hypervisor-level and privileged-level messages can be sent. Hypervisor-level messages are sent using the \texttt{msgsnd} instruction and cause hypervisor-level exceptions when received. Privileged-level messages are sent using the \texttt{msgsndp} instruction and cause privileged-level exceptions when received. For both 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 and MSR\textsubscript{EE}=1 or MSR\textsubscript{HV}=0 for the Directed Hypervisor Doorbell interrupt).

A Directed Privileged Doorbell exception remains until the corresponding interrupt occurs, or the exception is cleared by execution of a \texttt{mtspr}(DPDES) or \texttt{msgclr} instruction.

A Directed Hypervisor Doorbell exception remains until the corresponding interrupt occurs, or the exception is cleared by execution of a \texttt{msgclr} instruction.

If a doorbell exception is present and the corresponding interrupt is pended because MSR\textsubscript{EE}=0, additional doorbell exceptions are ignored until the exception is cleared.

10.3 Processor Control Registers

10.3.1 Directed Privileged Doorbell Exception State

The layout of the Directed Privileged Doorbell Exception State (DPDES) register is shown in Figure 85.

![Figure 85. 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 of an \texttt{msgsndp} or \texttt{msgclr} instruction or of modification of the DPDES register caused by execution of an \texttt{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, there is no similar register for the hypervisor. Privileged programs are able to read the DPDES in order to poll for Directed Privileged Doorbell exceptions when the corresponding interrupt is disabled (MSREE=1).
10.4 Processor Control Instructions

`msgsnd`, `msgsndp`, `msgclr`, and `msgclrp` instructions are provided for sending and clearing messages. `msgsyc` is provided to enable the thread that is target of a `msgsnd` instruction to ensure that stores performed by the message-sending thread before it executed `msgsnd` have been performed with respect to the target thread. `msgsndp` and `msgclrp` are privileged instructions, `msgsnd`, `msgclr`, and `msgsyc` are hypervisor privileged instructions.

**Message Send X-form**

```
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 PIR_{44:63} 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 10.2.

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, an `lw/sync` should be placed between the stores and the `msgsnd`. See Section 5.9.2.

```
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.
```

```
msgtype ← GPR(RB)_{32:36}
payload ← GPR(RB)_{37:63}
If(msgtype = 0x05)then
  send_msg(msgtype, payload)
```

`msgsnd` sends a message to other threads in the system. The message type and destination thread(s) are specified in RB.

**RB**

```
Type
0:31 Reserved
32:36
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.
```

**Figure 86. RB Contents for msgsnd**

The contents of RB are defined below. Bits 37:63 are referred to as the message payload.

<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>37:38</td>
<td>Broadcast</td>
</tr>
</tbody>
</table>

If `msgsnd` is used to notify the receiver that updates have been made to storage, an `lw/sync` should be placed between the stores and the `msgsnd`. See Section 5.9.2.

**Message Clear**

msgclr RB

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>238</th>
<th>RB</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>0</th>
</tr>
</thead>
</table>

\[ t \leftarrow \text{hypervisor thread number of executing thread} \]

If (msgtype = 0x05) then

- clear any Directed Hypervisor Doorbell exception for thread \( t \).

**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**

`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

**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**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>37:63</td>
<td><strong>TIRTAG</strong></td>
</tr>
</tbody>
</table>

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 10.2.

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 5.9.2.

**Figure 87. RB Contents for `msgsndp`**

The contents of RB are defined below. Bits 37:63 are referred to as the message payload.

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>37:56</td>
<td>Reserved</td>
</tr>
<tr>
<td>57:63</td>
<td><strong>TIRTAG</strong></td>
</tr>
</tbody>
</table>

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 10.2.

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 5.9.2.
**Message Clear Privileged X-form**

<table>
<thead>
<tr>
<th>msgclrp</th>
<th>RB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>31 (\text{///} 11 16 21)</td>
</tr>
</tbody>
</table>

\[\text{msgtype} \leftarrow (RB)_{32:36}\]
\[t \leftarrow \text{privileged thread number of executing thread}\]

\[
\text{IF}(\text{msgtype} = 0x05) \quad \text{then}
\]
\[\text{DPDES}_{63-t} \leftarrow 0\]

\text{msgclrp} clears a message previously accepted by the thread executing the \text{msgclrp}.

Let msgtype be \((RB)_{32:36}\), and let \(t\) be the privileged thread number of the thread executing the \text{msgclrp}.

If \(\text{msgtype} = 0x05\), then clear any Directed Privileged Doorbell exception that exists on thread \(t\) by setting \(\text{DPDES}_{63-t}\) to 0; otherwise, this instruction is treated as a no-op.

This instruction is privileged.

**Special Registers Altered:**

\[
\text{DPDES}
\]

---

**Programming Note**

\text{msgclrp} is typically issued only when MSR\(_{EE}=0\). If \text{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**

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

In conjunction with the \text{Synchronize} and \text{msgsnd} instructions, the \text{msgsync} instruction provides an ordering function for stores that have been performed with respect to the thread executing the \text{Synchronize} and \text{msgsnd} instructions, relative to data accesses by other threads that are performed after a Directed Hypervisor Doorbell interrupt has occurred, as described in the \text{Synchronize} instruction description on p. 1021.

This instruction is hypervisor privileged.

**Special Registers Altered:**

None

---

**Programming Note**

When used in conjunction with \text{msgsnd}, \text{Synchronize} with \(L = 0\) or 2 is executed on the thread that will execute the \text{msgsnd}, and \text{msgsync} is executed on another thread — typically the thread that is the target of the \text{msgsnd}, but possibly any other thread (partly because the software that services the Directed Hypervisor Doorbell interrupt may ultimately run on a thread other than that which received the exception). The \text{Synchronize} precedes the \text{msgsnd}, the \text{msgsync} is executed after the 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 \text{Synchronize}; see Section 5.9.2, "Synchronize Instruction".
Chapter 11. 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 6 (for data access) and Table 7 (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.

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 7, 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).

In situations such as context switch in which multiple SPRs are loaded in sequence, it is often the case that the composition of the implicit (implementation-specific, nonarchitectural) synchronizations performed for each individual mtspr will be excessive for the purpose. Software may identify such sequences by placing a mtgsr before the sequence. Hardware may respond to this identification by removing redundant synchronization so that the net synchronization effect approaches that of a single context synchronization at the end of the sequence. A potential side effect of the optimization is that the SPRs specified by the sequence may be loaded in an order other than that specified by the program with the result that an exception interrupts the sequence. mtspr instructions past the point of interruption may have loaded their SPRs. When control returns to the interrupted sequence, any such mtspr instructions are re-executed. The programmer must ensure that this side effect will not affect the outcome of the sequence. The degree of optimization is implementation-specific. Transaction failure may compromise optimization.
Because the individual mtsp<text class="highlighted">r</text> instructions in an optimized sequence may be executed in any order, a single sequence should not contain multiple loads of the same SPR, and should not contain any set of SPRs for which the relative order of execution of the mtsp<text class="highlighted">r</text> instructions targeting SPRs in the set matters.

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>event-based branch and rf&lt;text class=&quot;highlighted&quot;&gt;ebb&lt;/text&gt;</td>
<td>none</td>
<td>none</td>
<td>21</td>
</tr>
<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>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>mtsp&lt;text class=&quot;highlighted&gt;&quot;r&lt;/text&gt; (AMR)</td>
<td>CSI</td>
<td>CSI</td>
<td>13</td>
</tr>
<tr>
<td>mtsp&lt;text class=&quot;highlighted&gt;&quot;r&lt;/text&gt; (PIDR)</td>
<td>CSI</td>
<td>CSI</td>
<td>6</td>
</tr>
<tr>
<td>mtsp&lt;text class=&quot;highlighted&gt;&quot;r&lt;/text&gt; (DAWRn)</td>
<td>CSI</td>
<td>CSI</td>
<td></td>
</tr>
<tr>
<td>mtsp&lt;text class=&quot;highlighted&gt;&quot;r&lt;/text&gt; (DAWRXn)</td>
<td>CSI</td>
<td>CSI</td>
<td></td>
</tr>
<tr>
<td>mtsp&lt;text class=&quot;highlighted&gt;&quot;r&lt;/text&gt; (HRMOR)</td>
<td>CSI</td>
<td>CSI</td>
<td>11,17</td>
</tr>
<tr>
<td>mtsp&lt;text class=&quot;highlighted&gt;&quot;r&lt;/text&gt; (LPCR)</td>
<td>CSI</td>
<td>CSI</td>
<td>11, 14</td>
</tr>
<tr>
<td>mtsp&lt;text class=&quot;highlighted&gt;&quot;r&lt;/text&gt; (PTCR)</td>
<td>ptsync</td>
<td>CSI</td>
<td>3</td>
</tr>
<tr>
<td>mtmsrd&lt;text class=&quot;highlighted&gt;&quot;r&lt;/text&gt; (SF)</td>
<td>none</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>mtmsrd&lt;text class=&quot;highlighted&gt;&quot;r&lt;/text&gt; (TS)</td>
<td>none</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>mtmsrd&lt;text class=&quot;highlighted&gt;&quot;r&lt;/text&gt; (TM)</td>
<td>none</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>mtmsrd&lt;text class=&quot;highlighted&gt;&quot;r&lt;/text&gt;[d] (PR)</td>
<td>none</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>mtmsrd&lt;text class=&quot;highlighted&gt;&quot;r&lt;/text&gt;[d] (DR)</td>
<td>none</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>mtsp&lt;text class=&quot;highlighted&gt;&quot;r&lt;/text&gt; (LPIDR)</td>
<td>CSI</td>
<td>CSI</td>
<td>6</td>
</tr>
<tr>
<td>sibie</td>
<td>CSI</td>
<td>CSI</td>
<td>4</td>
</tr>
<tr>
<td>sibieg</td>
<td>CSI</td>
<td>CSI</td>
<td>4,6</td>
</tr>
<tr>
<td>sibia</td>
<td>CSI</td>
<td>CSI</td>
<td>4</td>
</tr>
<tr>
<td>sibmte</td>
<td>CSI</td>
<td>CSI</td>
<td>4,10</td>
</tr>
<tr>
<td>tblie</td>
<td>CSI</td>
<td>CSI</td>
<td>4,6</td>
</tr>
<tr>
<td>Store(PTE)</td>
<td>none</td>
<td>ptsync</td>
<td></td>
</tr>
<tr>
<td>Store(STE)</td>
<td>none</td>
<td>ptsync, CSI</td>
<td></td>
</tr>
<tr>
<td>Store(PRTE)</td>
<td>none</td>
<td>ptsync, CSI</td>
<td></td>
</tr>
<tr>
<td>Store(PATE)</td>
<td>none</td>
<td>ptsync, CSI</td>
<td></td>
</tr>
<tr>
<td>transaction failure and all TM instructions except tcheck</td>
<td>none</td>
<td>none</td>
<td>19</td>
</tr>
</tbody>
</table>

Table 6: 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>event-based branch and rfebb</td>
<td>none</td>
<td>none</td>
<td>21</td>
</tr>
<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>rscv</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>mtmsrd (SF)</td>
<td>none</td>
<td>none</td>
<td>7</td>
</tr>
<tr>
<td>mtmsrd (TS)</td>
<td>none</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>mtmsrd (TM)</td>
<td>none</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>mtmsr<a href="EE">d</a></td>
<td>none</td>
<td>none</td>
<td>1</td>
</tr>
<tr>
<td>mtmsr<a href="PR">d</a></td>
<td>none</td>
<td>none</td>
<td>8</td>
</tr>
<tr>
<td>mtmsr<a href="FP">d</a></td>
<td>none</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>mtmsr<a href="FE0,FE1">d</a></td>
<td>none</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>mtmsr<a href="TE">d</a></td>
<td>none</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>mtmsr<a href="IR">d</a></td>
<td>none</td>
<td>none</td>
<td>8</td>
</tr>
<tr>
<td>mtmsr<a href="RI">d</a></td>
<td>none</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>mtmsr(DEC)</td>
<td>none</td>
<td>none</td>
<td>9</td>
</tr>
<tr>
<td>mtmspr (PIDR)</td>
<td>CSI</td>
<td>CSI</td>
<td>6</td>
</tr>
<tr>
<td>mtmspr (IAMR)</td>
<td>none</td>
<td>CSI</td>
<td></td>
</tr>
<tr>
<td>mtmspr (TFHAR)</td>
<td>none</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>mtmspr (TEXASR)</td>
<td>none</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>mtmspr (CTRL)</td>
<td>none</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>mtmspr (FSCR)</td>
<td>none</td>
<td>CSI</td>
<td></td>
</tr>
<tr>
<td>mtmspr (DPDES)</td>
<td>none</td>
<td>CSI</td>
<td>17</td>
</tr>
<tr>
<td>mtmspr (CIABR)</td>
<td>none</td>
<td>CSI</td>
<td></td>
</tr>
<tr>
<td>mtmspr (HFSCR)</td>
<td>none</td>
<td>CSI</td>
<td></td>
</tr>
<tr>
<td>mtmspr (HDEC)</td>
<td>none</td>
<td>none</td>
<td>9</td>
</tr>
<tr>
<td>mtmspr (HRMOR)</td>
<td>none</td>
<td>CSI</td>
<td>8,11,17</td>
</tr>
<tr>
<td>mtmspr (LPCR)</td>
<td>none</td>
<td>CSI</td>
<td>11,12,14</td>
</tr>
<tr>
<td>mtmspr (LPIIDR)</td>
<td>CSI</td>
<td>CSI</td>
<td>6,14,17</td>
</tr>
<tr>
<td>mtmspr (PCR)</td>
<td>none</td>
<td>CSI</td>
<td>17</td>
</tr>
<tr>
<td>mtmspr (PTCR)</td>
<td>ptesync</td>
<td>CSI</td>
<td>3,17</td>
</tr>
<tr>
<td>mtmspr (Perf. Mon.)</td>
<td>none</td>
<td>CSI</td>
<td>15,18</td>
</tr>
<tr>
<td>mtmspr (BESCR)</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>silbieg</td>
<td>none</td>
<td>CSI</td>
<td>4,6</td>
</tr>
<tr>
<td>silbia</td>
<td>none</td>
<td>CSI</td>
<td>4</td>
</tr>
<tr>
<td>slbmtle</td>
<td>none</td>
<td>CSI</td>
<td>4,8,10</td>
</tr>
<tr>
<td>tblie</td>
<td>none</td>
<td>CSI</td>
<td>4,6</td>
</tr>
<tr>
<td>tbliel</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>
</tbody>
</table>

Table 7: Synchronization requirements for instruction fetch and/or execution

Transaction failure and all TM instructions except *tcheck*.

Table 7: Synchronization requirements for instruction fetch and/or execution
Notes:

1. The effect of changing the EE bit is immediate, even if the \texttt{mtmsr[d]} instruction is not context synchronizing (i.e., even if \( L = 1 \)).
   - If an \texttt{mtmsr[d]} instruction sets the EE bit to 0, neither an External interrupt, a Decrementer interrupt nor a Performance Monitor interrupt occurs after the \texttt{mtmsr[d]} is executed.
   - If an \texttt{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 \texttt{mtmsr[d]} is executed, and before the next instruction is executed in the program that set EE to 1.
   - If a hypervisor executes the \texttt{mtmsr[d]} instruction that sets the EE bit to 0, a Hypervisor Decrementer interrupt does not occur after \texttt{mtmsr[d]} is executed as long as the thread remains in hypervisor state.
   - If the hypervisor executes an \texttt{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 \texttt{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 real addressing mode. Modifying the PTCR requires that the following conditions be achieved on all threads on the processor:
   - the thread is in hypervisor real addressing mode
   - all previous accesses (implicit and explicit) initiated when the thread was not in hypervisor 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 \texttt{slibie}, \texttt{slibieg}, \texttt{slibia}, \texttt{slibmte}, \texttt{tlbie}, or \texttt{tlbie} 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 \texttt{slibie}, \texttt{slibieg}, \texttt{slibia}, \texttt{slibmte}, \texttt{tlbie} or \texttt{tlbie} instruction ensures that storage accesses associated with instructions following the context synchronizing instruction will not use the TLB entry(s) being invalidated.

   (For \texttt{tlbie} and \texttt{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 \texttt{ptesync} instruction must also be used, either before or after the \texttt{tlbie} or \texttt{tlbie} instruction. These effects of the \texttt{ptesync} instruction are described in the last paragraph of Note 5.)

5. The notation “\texttt{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 \texttt{Store} instruction because (a) stores are not performed out-of-order and (b) address translations associated with instructions preceding the \texttt{Store} instruction are not performed again after the store has been performed (see Section 5.5). These properties ensure that all address translations associated with instructions preceding the \texttt{Store} instruction will be performed using the old contents of the PTE.

   The \texttt{ptesync} instruction after the \texttt{Store} instruction ensures that all searches of the Page Table that are performed after the \texttt{ptesync} instruction completes will use the value stored (or a value stored subsequently). The context synchronizing instruction after the \texttt{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 after the \texttt{Store} instructions are performed with respect to that thread or mechanism.

   The \texttt{ptesync} instruction also ensures that all storage accesses associated with instructions preceding the \texttt{ptesync} instruction, and 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, 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 \texttt{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 invalidations have completed and that all side effects of the invalidations have taken effect).
Section 5.10 gives examples of using \texttt{tlbie}, \texttt{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 5.10 are satisfied, the \texttt{lwsync} 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\textsubscript{SF} from 1 to 0, the \texttt{mtmsrd} instruction must have an effective address that is less than \(2^{32} - 4\). Furthermore, when changing MSR\textsubscript{SF} from 0 to 1, the \texttt{mtmsrd} instruction must not be at effective address \(2^{32} - 4\) (see Section 5.3.2 on page 981).

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\textsubscript{IR} to 1 early in an operating system interrupt handler, advantage can sometimes be taken of the fact that EA\textsubscript{0:3} are ignored when forming the real address when address translation is disabled and MSR\textsubscript{HV} = 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.

```assembly
la    rx,target
li    ry,0x000
sldi  ry,ry,48
or    rx,rx,ry  # set high-order
       nibble of target
       addr to 0xn
mtctr rx
bcctr  # branch to targ
targ:  mfmsr rx
orir  x,rz,0x0020
mtmsrd rx  # set MSR\textsubscript{IR} to 1
```

The \texttt{mtmsrd} does not cause an implicit branch in real address space because the real address of the next sequential instruction is independent of MSR\textsubscript{IR}. Using \texttt{mtmsrd}, rather than \texttt{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 \texttt{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 \texttt{slbmte} instruction alters the mapping, or associated attributes, of a currently mapped ESID, the \texttt{slbmte} must be preceded by an \texttt{slbie} (or \texttt{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 \texttt{slbie} and the \texttt{slbmte}, 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 \texttt{slbmte}.
No `slbie` (or `slbia`) is needed if the `slbmte` 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 HRMOR or the VC field of the LPCR is modified, software must invalidate all implementation-specific lookaside information used in address translation that depends on the old contents of the register or field (i.e., the contents immediately before the modification). The `slbia` 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\textsubscript{MER} = 1 does not necessarily ensure that the exception effects of LPCR\textsubscript{MER} are consistent with the contents of LPCR\textsubscript{MER}. 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 LPCR\textsubscript{HR} must not be altered when MSR\textsubscript{DR}=1 or MSR\textsubscript{IR}=1; if they are, the results are undefined.

---

**Programming Note**

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. (The tables used for translation are determined by the partition ID and LPCR\textsubscript{HR} is used as a shortcut. See Section 5.7.6 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. These instructions and events, with the exception of nested `begin`, nested `tend`, TM instructions that except or are described to be treated as no-ops, Transaction Abort Conditional instructions that do not abort, and events and `rfebb` instructions for which the event did not take place in Transactional state, will change MSR\textsubscript{TS}. No software synchronization is required.
Appendix A. 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.

1, 5, 6

The following primary opcodes have unused extended opcodes. Their unused extended opcodes can be determined from the opcode maps in Appendix C of Book Appendices. All unused extended opcodes are illegal.

4, 19, 30, 31, 56, 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.

<table>
<thead>
<tr>
<th>primary / extended opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 / 0b10110_000001</td>
</tr>
<tr>
<td>4 / 0b11110_000001</td>
</tr>
<tr>
<td>4 / 0b11000_000010</td>
</tr>
<tr>
<td>60 / 0b01011_01000.</td>
</tr>
<tr>
<td>60 / 0b10101_1011.</td>
</tr>
<tr>
<td>60 / 0b11101_1011.</td>
</tr>
<tr>
<td>63 / 0b10010_00100.</td>
</tr>
<tr>
<td>63 / 0b11010_00100.</td>
</tr>
<tr>
<td>63 / 0b10010_00111.</td>
</tr>
</tbody>
</table>

An instruction consisting entirely of binary 0s is illegal, and is guaranteed to be illegal in all future versions of this architecture.
Appendix B. 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 1.8.2, "Illegal Instruction Class" on page 22) and the extended opcode shown below.

   256  Service Processor “Attention”

2. Instructions for the POWER Architecture that have not been included in the Power ISA.

3. Implementation-specific instructions used to conform to the Power ISA specification.

4. Any other implementation-dependent instructions that are not defined in the Power ISA.
Appendix C. Opcode Maps

This appendix contains opcode maps showing the primary opcodes, extended opcodes, and expanded opcodes.

Table 8 describes the conventions used in the opcode maps.

The instruction consisting entirely of binary 0s causes the system illegal instruction error handler to be invoked for all members of the POWER family, and this is likely to remain true in future models (it is guaranteed in the Power ISA). An instruction having primary opcode 0 but not consisting entirely of binary 0s is reserved except for the following extended opcode (instruction bits 21:30).

256 Service Processor “Attention”

Table 8: 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>primary opcode (decimal format)</td>
</tr>
<tr>
<td>xop</td>
<td></td>
<td></td>
<td>extended or expanded opcode image (binary format)</td>
</tr>
</tbody>
</table>

- 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 |
|          |         |           | H: hypervisor-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 15: EXT62: Extended Opcode Map for Primary Opcode 62 (opcode bits 21:30)

<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>sub</td>
<td>tdi</td>
<td>twi</td>
<td>ext</td>
<td>addi</td>
<td>addic</td>
<td>add</td>
<td>addis</td>
</tr>
<tr>
<td>cmpli</td>
<td>cmpli</td>
<td>addi</td>
<td>addic</td>
<td>add</td>
<td>addis</td>
<td>std</td>
<td>std</td>
</tr>
<tr>
<td>bc[[l][a]]</td>
<td>EXT17</td>
<td>b[[l][a]]</td>
<td>EXT19</td>
<td>rliwm[.]</td>
<td>rliwm[.]</td>
<td>EXT30</td>
<td>EXT31</td>
</tr>
<tr>
<td>ori</td>
<td>oris</td>
<td>xori</td>
<td>xoris</td>
<td>andi.</td>
<td>andis.</td>
<td>EXT57</td>
<td>EXT58</td>
</tr>
<tr>
<td>100</td>
<td>lwz</td>
<td>lwzu</td>
<td>lbz</td>
<td>lbzu</td>
<td>stw</td>
<td>stwu</td>
<td>stb</td>
</tr>
<tr>
<td>101</td>
<td>lhz</td>
<td>lhz</td>
<td>lha</td>
<td>lha</td>
<td>sflw</td>
<td>sflwu</td>
<td>stlw</td>
</tr>
<tr>
<td>110</td>
<td>lfs</td>
<td>lfs</td>
<td>lfd</td>
<td>lfd</td>
<td>sfs</td>
<td>sfsu</td>
<td>std</td>
</tr>
<tr>
<td>111</td>
<td>liq</td>
<td>liq</td>
<td>EXT57</td>
<td>EXT58</td>
<td>EXT59</td>
<td>EXT60</td>
<td>EXT61</td>
</tr>
</tbody>
</table>

### Table 10: EXT17: Extended Opcode Map for Primary Opcode 17 (opcode bits 30:31)

<table>
<thead>
<tr>
<th>00</th>
<th>01</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>scv</td>
<td>scv</td>
<td>scv</td>
<td>scv</td>
</tr>
</tbody>
</table>

### Table 11: EXT30: Extended Opcode Map for Primary Opcode 30 (opcode bits 27:30)

<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>rldic[.]</td>
<td>rldic[.]</td>
<td>rldic[.]</td>
<td>rldic[.]</td>
<td>rldic[.]</td>
<td>rldic[.]</td>
<td>rldic[.]</td>
<td>rldic[.]</td>
</tr>
<tr>
<td>0</td>
<td>P1</td>
<td>P1</td>
<td>P1</td>
<td>P1</td>
<td>P1</td>
<td>P1</td>
<td>P1</td>
</tr>
<tr>
<td>1</td>
<td>M0</td>
<td>M0</td>
<td>M0</td>
<td>M0</td>
<td>M0</td>
<td>M0</td>
<td>M0</td>
</tr>
</tbody>
</table>

### Table 12: EXT57: Extended Opcode Map for Primary Opcode 57 (opcode bits 30:31)

<table>
<thead>
<tr>
<th>00</th>
<th>01</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>ldp</td>
<td>ldp</td>
<td>ldp</td>
<td>ldp</td>
</tr>
</tbody>
</table>

### Table 13: EXT58: Extended Opcode Map for Primary Opcode 58 (opcode bits 30:31)

<table>
<thead>
<tr>
<th>00</th>
<th>01</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>lfd</td>
<td>lfd</td>
<td>lfd</td>
<td>lfd</td>
</tr>
</tbody>
</table>

### Table 14: EXT61: Extended Opcode Map for Primary Opcode 61 (opcode bits 21:30)

<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>stfdp</td>
<td>stfdp</td>
<td>stfv</td>
<td>stxsd</td>
<td>stxssp</td>
<td>stfdp</td>
<td>stfdp</td>
<td>stfdp</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

### Table 15: EXT62: Extended Opcode Map for Primary Opcode 62 (opcode bits 21:30)

<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>std</td>
<td>std</td>
<td>std</td>
<td>std</td>
<td>std</td>
<td>std</td>
<td>std</td>
<td>std</td>
</tr>
</tbody>
</table>
Table 16: EXT04: Extended Opcode Map for Primary Opcode 4 (opcode bits 0:5) (Sheet 1 of 8)

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>vaddubm</td>
<td>v2.03</td>
</tr>
<tr>
<td>00001</td>
<td>vmul10cuq</td>
<td>v2.03</td>
</tr>
<tr>
<td>00010</td>
<td>vadduw</td>
<td>v2.03</td>
</tr>
<tr>
<td>00011</td>
<td>vadduq</td>
<td>v2.03</td>
</tr>
<tr>
<td>00100</td>
<td>vaddcuq</td>
<td>v2.03</td>
</tr>
<tr>
<td>00101</td>
<td>vaddcuw</td>
<td>v2.03</td>
</tr>
<tr>
<td>00110</td>
<td>vaddcuw</td>
<td>v2.03</td>
</tr>
<tr>
<td>01000</td>
<td>vaddubs</td>
<td>v2.03</td>
</tr>
<tr>
<td>01001</td>
<td>vmul10cuq</td>
<td>v2.03</td>
</tr>
<tr>
<td>01010</td>
<td>vadduw</td>
<td>v2.03</td>
</tr>
<tr>
<td>01011</td>
<td>vadduw</td>
<td>v2.03</td>
</tr>
<tr>
<td>01100</td>
<td>vaddubs</td>
<td>v2.03</td>
</tr>
<tr>
<td>01101</td>
<td>bcdcpsgn.</td>
<td>v2.03</td>
</tr>
<tr>
<td>01110</td>
<td>vadduw</td>
<td>v2.03</td>
</tr>
<tr>
<td>10000</td>
<td>vsububm</td>
<td>v2.03</td>
</tr>
<tr>
<td>10001</td>
<td>bcdadd.</td>
<td>v2.03</td>
</tr>
<tr>
<td>10010</td>
<td>vsububm</td>
<td>v2.03</td>
</tr>
<tr>
<td>10011</td>
<td>bcdus.</td>
<td>v2.03</td>
</tr>
<tr>
<td>10100</td>
<td>bcdus.</td>
<td>v2.03</td>
</tr>
<tr>
<td>10101</td>
<td>bcdus.</td>
<td>v2.03</td>
</tr>
<tr>
<td>10110</td>
<td>vsububm</td>
<td>v2.03</td>
</tr>
<tr>
<td>11000</td>
<td>bcdadd.</td>
<td>v2.03</td>
</tr>
<tr>
<td>11001</td>
<td>bcdadd.</td>
<td>v2.03</td>
</tr>
<tr>
<td>11010</td>
<td>vsububm</td>
<td>v2.03</td>
</tr>
<tr>
<td>11011</td>
<td>vsububm</td>
<td>v2.03</td>
</tr>
<tr>
<td>11100</td>
<td>bcdadd.</td>
<td>v2.03</td>
</tr>
<tr>
<td>11101</td>
<td>bcdadd.</td>
<td>v2.03</td>
</tr>
</tbody>
</table>

Appendix C. Opcode Maps
Table 16: EXT04: Extended Opcode Map for Primary Opcode 4 (opcode bits 0:5) (Sheet 2 of 8)

<table>
<thead>
<tr>
<th>Opcode</th>
<th>001000</th>
<th>001001</th>
<th>001010</th>
<th>001011</th>
<th>001100</th>
<th>001101</th>
<th>001110</th>
<th>001111</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>vsum2w</td>
<td>vsum2s</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>00010</td>
</tr>
<tr>
<td>vsum4ubs</td>
<td>vsum4sbs</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>00100</td>
</tr>
<tr>
<td>vmulpsb</td>
<td>vmulosh</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>00110</td>
</tr>
<tr>
<td>vmulosw</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>01000</td>
</tr>
<tr>
<td>vsum4shs</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>01100</td>
</tr>
<tr>
<td>vsum4sbs</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>01110</td>
</tr>
<tr>
<td>vsum4shs</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>10000</td>
</tr>
<tr>
<td>vpmusw</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>10010</td>
</tr>
<tr>
<td>vpmusm</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>10100</td>
</tr>
<tr>
<td>vpmusm</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>10110</td>
</tr>
<tr>
<td>vpmusw</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>11000</td>
</tr>
<tr>
<td>vpmusmb</td>
<td>vpmusm</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>11010</td>
</tr>
<tr>
<td>vpmusmb</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>11100</td>
</tr>
<tr>
<td>vpmusmb</td>
<td>vpmusm</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>11110</td>
</tr>
</tbody>
</table>

Note: The table continues with similar entries for each opcode combination, listing various operations such as multiplication, division, and logical operations.
Table 16: EXT04: Extended Opcode Map for Primary Opcode 4 (opcode bits 0:5) (Sheet 3 of 8)

|   | 00000 | 00001 | 00010 | 00011 | 00100 | 00101 | 00110 | 00111 | 01000 | 01001 | 01010 | 01011 | 01100 | 01101 | 01110 | 01111 | 10000 | 10001 | 10010 | 10011 | 10100 | 10101 | 10110 | 10111 | 11000 | 11001 | 11010 | 11011 | 11100 | 11101 | 11110 | 11111 |
|---|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|
|   |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |
Table 16: EXT04: Extended Opcode Map for Primary Opcode 4 (opcode bits 0:5) (Sheet 4 of 8)

<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></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 continued...
Table 16: EXT04: Extended Opcode Map for Primary Opcode 4 (opcode bits 0:5) (Sheet 5 of 8)

<table>
<thead>
<tr>
<th>Opcode Bits</th>
<th>vmhaddhs</th>
<th>vmhaddhs</th>
<th>vmladduhm</th>
<th>vmsumudm</th>
<th>vmsumubm</th>
<th>vmsummbm</th>
<th>vmsumuhs</th>
<th>vmsumuhs</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td></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>
<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 16: EXT04: Extended Opcode Map for Primary Opcode 4 (opcode bits 0:5) (Sheet 6 of 8)
### Table 16: EXT04: Extended Opcode Map for Primary Opcode 4 (opcode bits 0:5) (Sheet 7 of 8)

<table>
<thead>
<tr>
<th>Opcode</th>
<th>maddhd</th>
<th>maddhdu</th>
<th>maddld</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00001</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00010</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00011</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00100</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00101</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00110</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00111</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01001</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01010</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01011</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01100</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01101</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01110</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01111</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10001</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10010</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10011</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10100</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10101</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10110</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10111</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11001</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11010</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11011</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11100</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11101</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11110</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11111</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 16: EXT04: Extended Opcode Map for Primary Opcode 4 (opcode bits 0:5) (Sheet 8 of 8)

<table>
<thead>
<tr>
<th></th>
<th>111000</th>
<th>111001</th>
<th>111010</th>
<th>111011</th>
<th>111100</th>
<th>111101</th>
<th>111110</th>
<th>111111</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>
<tr>
<td></td>
<td>111000</td>
<td>111001</td>
<td>111010</td>
<td>111011</td>
<td>111100</td>
<td>111101</td>
<td>111110</td>
<td>111111</td>
</tr>
</tbody>
</table>
### Table 17: XPND04-1A: Extended Opcode Map for PO=4 XO=0b10110 _000001 (opcode bits 11:15)

<table>
<thead>
<tr>
<th>Opcode Bits</th>
<th>Opcode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 00 000</td>
<td>bcdctsq.</td>
<td>VX</td>
</tr>
<tr>
<td>0 00 001</td>
<td>bcdctsq.</td>
<td>VX</td>
</tr>
<tr>
<td>0 00 010</td>
<td>bcdctsq.</td>
<td>VX</td>
</tr>
<tr>
<td>0 00 011</td>
<td>bcdctsq.</td>
<td>VX</td>
</tr>
<tr>
<td>0 00 100</td>
<td>bcdctz.</td>
<td>VX</td>
</tr>
<tr>
<td>0 00 101</td>
<td>bcdctn.</td>
<td>VX</td>
</tr>
<tr>
<td>0 00 110</td>
<td>bcdcfz.</td>
<td>VX</td>
</tr>
<tr>
<td>0 00 111</td>
<td>bcdcfn.</td>
<td>VX</td>
</tr>
<tr>
<td>0 01 000</td>
<td>vclzlsbb</td>
<td>v3.0 VX</td>
</tr>
<tr>
<td>0 01 001</td>
<td>vctzlsbb</td>
<td>v3.0 VX</td>
</tr>
<tr>
<td>0 01 010</td>
<td>vnegw</td>
<td>v3.0 VX</td>
</tr>
<tr>
<td>0 01 011</td>
<td>vnegd</td>
<td>v3.0 VX</td>
</tr>
<tr>
<td>0 01 100</td>
<td>vextsb2w</td>
<td>v3.0 VX</td>
</tr>
<tr>
<td>0 01 101</td>
<td>vextsh2w</td>
<td>v3.0 VX</td>
</tr>
<tr>
<td>0 01 110</td>
<td>vextsb2d</td>
<td>v3.0 VX</td>
</tr>
<tr>
<td>0 01 111</td>
<td>vextsh2d</td>
<td>v3.0 VX</td>
</tr>
<tr>
<td>0 10 000</td>
<td>vextsw2d</td>
<td>v3.0 VX</td>
</tr>
<tr>
<td>0 10 001</td>
<td>vctzb</td>
<td>v3.0 VX</td>
</tr>
<tr>
<td>0 10 010</td>
<td>vctzh</td>
<td>v3.0 VX</td>
</tr>
<tr>
<td>0 10 011</td>
<td>vctzw</td>
<td>v3.0 VX</td>
</tr>
<tr>
<td>0 10 100</td>
<td>vctzd</td>
<td>v3.0 VX</td>
</tr>
</tbody>
</table>

### Table 18: XPND04-1B: Extended Opcode Map for PO=4 XO=0b11110 _000001 (opcode bits 11:15)

<table>
<thead>
<tr>
<th>Opcode Bits</th>
<th>Opcode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 00 000</td>
<td>bcdctsq.</td>
<td>VX</td>
</tr>
<tr>
<td>0 00 001</td>
<td>bcdctsq.</td>
<td>VX</td>
</tr>
<tr>
<td>0 00 010</td>
<td>bcdctsq.</td>
<td>VX</td>
</tr>
<tr>
<td>0 00 011</td>
<td>bcdctsq.</td>
<td>VX</td>
</tr>
<tr>
<td>0 00 100</td>
<td>bcdctz.</td>
<td>VX</td>
</tr>
<tr>
<td>0 00 101</td>
<td>bcdctn.</td>
<td>VX</td>
</tr>
<tr>
<td>0 00 110</td>
<td>bcdcfz.</td>
<td>VX</td>
</tr>
<tr>
<td>0 00 111</td>
<td>bcdcfn.</td>
<td>VX</td>
</tr>
<tr>
<td>0 01 000</td>
<td>vclzlsbb</td>
<td>v3.0 VX</td>
</tr>
<tr>
<td>0 01 001</td>
<td>vctzlsbb</td>
<td>v3.0 VX</td>
</tr>
<tr>
<td>0 01 010</td>
<td>vnegw</td>
<td>v3.0 VX</td>
</tr>
<tr>
<td>0 01 011</td>
<td>vnegd</td>
<td>v3.0 VX</td>
</tr>
<tr>
<td>0 01 100</td>
<td>vextsb2w</td>
<td>v3.0 VX</td>
</tr>
<tr>
<td>0 01 101</td>
<td>vextsh2w</td>
<td>v3.0 VX</td>
</tr>
<tr>
<td>0 01 110</td>
<td>vextsb2d</td>
<td>v3.0 VX</td>
</tr>
<tr>
<td>0 01 111</td>
<td>vextsh2d</td>
<td>v3.0 VX</td>
</tr>
<tr>
<td>0 10 000</td>
<td>vextsw2d</td>
<td>v3.0 VX</td>
</tr>
<tr>
<td>0 10 001</td>
<td>vctzb</td>
<td>v3.0 VX</td>
</tr>
<tr>
<td>0 10 010</td>
<td>vctzh</td>
<td>v3.0 VX</td>
</tr>
<tr>
<td>0 10 011</td>
<td>vctzw</td>
<td>v3.0 VX</td>
</tr>
<tr>
<td>0 10 100</td>
<td>vctzd</td>
<td>v3.0 VX</td>
</tr>
</tbody>
</table>

### Table 19: XPND04-2: Extended Opcode Map for PO=4 XO=0b11000 000010 (opcode bits 11:15)

<table>
<thead>
<tr>
<th>Opcode Bits</th>
<th>Opcode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 00 000</td>
<td>vclzlsbb</td>
<td>v3.0 VX</td>
</tr>
<tr>
<td>0 00 001</td>
<td>vctzlsbb</td>
<td>v3.0 VX</td>
</tr>
<tr>
<td>0 00 010</td>
<td>vnegw</td>
<td>v3.0 VX</td>
</tr>
<tr>
<td>0 00 011</td>
<td>vnegd</td>
<td>v3.0 VX</td>
</tr>
<tr>
<td>0 00 100</td>
<td>vextsb2w</td>
<td>v3.0 VX</td>
</tr>
<tr>
<td>0 00 101</td>
<td>vextsh2w</td>
<td>v3.0 VX</td>
</tr>
<tr>
<td>0 00 110</td>
<td>vextsb2d</td>
<td>v3.0 VX</td>
</tr>
<tr>
<td>0 00 111</td>
<td>vextsh2d</td>
<td>v3.0 VX</td>
</tr>
<tr>
<td>0 01 000</td>
<td>vextsw2d</td>
<td>v3.0 VX</td>
</tr>
<tr>
<td>0 01 001</td>
<td>vctzb</td>
<td>v3.0 VX</td>
</tr>
<tr>
<td>0 01 010</td>
<td>vctzh</td>
<td>v3.0 VX</td>
</tr>
<tr>
<td>0 01 011</td>
<td>vctzw</td>
<td>v3.0 VX</td>
</tr>
<tr>
<td>0 01 100</td>
<td>vctzd</td>
<td>v3.0 VX</td>
</tr>
<tr>
<td>0 10 000</td>
<td>vextsb2w</td>
<td>v3.0 VX</td>
</tr>
<tr>
<td>0 10 001</td>
<td>vextsh2w</td>
<td>v3.0 VX</td>
</tr>
<tr>
<td>0 10 010</td>
<td>vextsb2d</td>
<td>v3.0 VX</td>
</tr>
<tr>
<td>0 10 011</td>
<td>vextsh2d</td>
<td>v3.0 VX</td>
</tr>
<tr>
<td>0 10 100</td>
<td>vextsw2d</td>
<td>v3.0 VX</td>
</tr>
<tr>
<td>0 10 101</td>
<td>vctzb</td>
<td>v3.0 VX</td>
</tr>
<tr>
<td>0 10 110</td>
<td>vctzh</td>
<td>v3.0 VX</td>
</tr>
<tr>
<td>0 10 111</td>
<td>vctzw</td>
<td>v3.0 VX</td>
</tr>
<tr>
<td>0 11 000</td>
<td>vctzd</td>
<td>v3.0 VX</td>
</tr>
</tbody>
</table>
Table 20: EXT19: Extended Opcode Map for Primary Opcode 19 (opcode bits 21:30) (Sheet 1 of 4)

<table>
<thead>
<tr>
<th>Opcode Bits</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>mcff</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00001</td>
<td></td>
<td>crnor</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>crnand</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00101</td>
<td></td>
<td>crorc</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>creqv</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>

* P1 = Primary register
* XL = Extended Longword
* mcrf = move condition register to floating point
* addpis = add PC with immediate
* crnor = condition register nor
* crnand = condition register nand
* crorc = condition register or complement
* crorc = condition register or
* creqv = condition register equiv
* crand = condition register and
* crandc = condition register and complement
<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>00000</td>
<td></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>
<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 20: EXT19: Extended Opcode Map for Primary Opcode 19 (opcode 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>00000</td>
<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>
<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>10000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10001</td>
<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>
<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>

1158 Power ISA™ Appendices
### Table 20: EXT19: Extended Opcode Map for Primary Opcode 19 (opcode bits 21:30) (Sheet 4 of 4)

<table>
<thead>
<tr>
<th>00000</th>
<th>00001</th>
<th>00010</th>
<th>00011</th>
<th>00100</th>
<th>00101</th>
<th>00110</th>
<th>00111</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>
<th>10000</th>
<th>10001</th>
<th>10010</th>
<th>10011</th>
<th>10100</th>
<th>10101</th>
<th>10110</th>
<th>10111</th>
<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></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>
<td></td>
</tr>
<tr>
<td>00001</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>
<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>
<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>00011</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>
<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>
<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>00101</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>
<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>
<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>00111</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>
<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>
<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>01001</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>
<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>
<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>01011</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>
<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>
<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>01101</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>
<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>
<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>01111</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>
<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>
<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>10001</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>
<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>
<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>10011</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>
<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>
<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>10101</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>
<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>
<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>10111</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>
<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>
<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>11001</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>
<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>
<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>11011</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>
<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>
<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>11101</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>
<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>
<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>11111</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>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 21: EXT31: Extended Opcode Map for Primary Opcode 31 (opcode bits 21:30) (Sheet 1 of 4)

<table>
<thead>
<tr>
<th>Opcode Bits</th>
<th>Description</th>
<th>Function</th>
<th>Usage Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>cmpl</td>
<td>P1 X</td>
<td></td>
</tr>
<tr>
<td>00001</td>
<td>cmp</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>00001</td>
<td>cmpl</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>00100</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00101</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00101</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00110</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00111</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10000</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10001</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10010</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10010</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10100</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10100</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10101</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10101</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10110</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10110</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11000</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11000</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11010</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11010</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11100</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11100</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11110</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11110</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Notes:**
- Opcodes 00000-00011 Reserved.
- Opcodes 00100-00111 Reserved.
- Opcodes 10000-11111 Reserved.

**Function Codes:**
- cmpl: Compare Immediate
- cmp: Compare
- Reserved: Not defined.

**Usage Notes:**
- P1: First operand is a general-purpose register.
- X: Second operand is a general-purpose register.
- Reserved: Not defined.

**Function Code Details:**
- copy: Copy
- paste: Paste
- cpabort: Coprocessor abort
- paste[.]
<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
<th>Symbol</th>
<th>W</th>
<th>X</th>
<th>Y</th>
<th>Z</th>
<th>Instruction</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>subf()</td>
<td>XO</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>lxswax</td>
<td>isel A</td>
</tr>
<tr>
<td>00010</td>
<td>subf()</td>
<td>PPC</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>lxswax</td>
<td>X</td>
</tr>
<tr>
<td>00100</td>
<td>mulhd()</td>
<td>XO</td>
<td>0.06</td>
<td>0.06</td>
<td>0.06</td>
<td>0.06</td>
<td>lxswax</td>
<td>X</td>
</tr>
<tr>
<td>00110</td>
<td>negate()</td>
<td>XO</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>lxswax</td>
<td>X</td>
</tr>
<tr>
<td>01000</td>
<td>subf()</td>
<td>XO</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>stxswax</td>
<td>X</td>
</tr>
<tr>
<td>01010</td>
<td>negate()</td>
<td>XO</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>stxswax</td>
<td>X</td>
</tr>
<tr>
<td>01100</td>
<td>negate()</td>
<td>XO</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>stxswax</td>
<td>X</td>
</tr>
<tr>
<td>10000</td>
<td>subfco()</td>
<td>XO</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>lxsspx</td>
<td>X</td>
</tr>
<tr>
<td>10010</td>
<td>subfco()</td>
<td>PPC</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>lxsspx</td>
<td>X</td>
</tr>
<tr>
<td>10100</td>
<td>mulhd()</td>
<td>XO</td>
<td>0.06</td>
<td>0.06</td>
<td>0.06</td>
<td>0.06</td>
<td>lxssdx</td>
<td>X</td>
</tr>
<tr>
<td>10110</td>
<td>negate()</td>
<td>XO</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>lxssdx</td>
<td>X</td>
</tr>
<tr>
<td>11000</td>
<td>modsd</td>
<td>XO</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>lxswax</td>
<td>X</td>
</tr>
<tr>
<td>11010</td>
<td>addo()</td>
<td>XO</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>lxswax</td>
<td>X</td>
</tr>
<tr>
<td>11100</td>
<td>divdeo()</td>
<td>XO</td>
<td>0.06</td>
<td>0.06</td>
<td>0.06</td>
<td>0.06</td>
<td>lxswax</td>
<td>X</td>
</tr>
<tr>
<td>11110</td>
<td>divdeo()</td>
<td>XO</td>
<td>0.06</td>
<td>0.06</td>
<td>0.06</td>
<td>0.06</td>
<td>lxswax</td>
<td>X</td>
</tr>
</tbody>
</table>

*Table 21: EXT31: Extended Opcode Map for Primary Opcode 31 (opcode bits 21:30) (Sheet 2 of 4)*
<table>
<thead>
<tr>
<th>Code</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>
Table 21: EXT31: Extended Opcode Map for Primary Opcode 31 (opcode bits 21:30) (Sheet 4 of 4)

<table>
<thead>
<tr>
<th></th>
<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>0000</td>
<td>slw[,]</td>
<td>cntlw[,]</td>
<td>sid[,]</td>
<td>P1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>0001</td>
<td>P1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>0010</td>
<td>P1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>0011</td>
<td>P1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>0100</td>
<td>P1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>0101</td>
<td>P1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>0110</td>
<td>P1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>0111</td>
<td>P1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>1000</td>
<td>P1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>1001</td>
<td>P1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>1010</td>
<td>P1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>1011</td>
<td>P1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>1100</td>
<td>P1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>1101</td>
<td>P1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>1110</td>
<td>P1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>1111</td>
<td>P1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>
Table 22: EXT59: Extended Opcode Map for Primary Opcode 59 (opcode bits 21:30) (Sheet 1 of 4)

<table>
<thead>
<tr>
<th>Opcode Bits</th>
<th>Opcode</th>
<th>Function</th>
<th>Version 2.05</th>
<th>Version 3.0 X</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>dadd</td>
<td>X</td>
<td>Z23</td>
<td></td>
</tr>
<tr>
<td>00001</td>
<td>dimul</td>
<td>X</td>
<td>Z23</td>
<td></td>
</tr>
<tr>
<td>00010</td>
<td>dscalar</td>
<td>X</td>
<td>Z23</td>
<td></td>
</tr>
<tr>
<td>00011</td>
<td>dcmulx</td>
<td>X</td>
<td>Z23</td>
<td></td>
</tr>
<tr>
<td>00100</td>
<td>dcmpo</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00101</td>
<td>dtsrex</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00110</td>
<td>dtstdc</td>
<td>Z22</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00111</td>
<td>dtstdg</td>
<td>Z22</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01000</td>
<td>dctdp</td>
<td>X</td>
<td>Z23</td>
<td></td>
</tr>
<tr>
<td>01001</td>
<td>dctfxx</td>
<td>X</td>
<td>Z23</td>
<td></td>
</tr>
<tr>
<td>01010</td>
<td>ddsipd</td>
<td>X</td>
<td>Z23</td>
<td></td>
</tr>
<tr>
<td>01011</td>
<td>dxes</td>
<td>X</td>
<td>Z23</td>
<td></td>
</tr>
<tr>
<td>01100</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01101</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01110</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01111</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10000</td>
<td>dsub</td>
<td>X</td>
<td>Z23</td>
<td></td>
</tr>
<tr>
<td>10001</td>
<td>div</td>
<td>X</td>
<td>Z23</td>
<td></td>
</tr>
<tr>
<td>10010</td>
<td>dscalr</td>
<td>X</td>
<td>Z23</td>
<td></td>
</tr>
<tr>
<td>10011</td>
<td>dcmplx</td>
<td>X</td>
<td>Z23</td>
<td></td>
</tr>
<tr>
<td>10100</td>
<td>drgb</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10101</td>
<td>dtstaf</td>
<td>Z22</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10110</td>
<td>dtstdc</td>
<td>Z22</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10111</td>
<td>dtstdg</td>
<td>Z22</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11000</td>
<td>drsp</td>
<td>X</td>
<td>Z23</td>
<td></td>
</tr>
<tr>
<td>11001</td>
<td>dctfxx</td>
<td>X</td>
<td>Z23</td>
<td></td>
</tr>
<tr>
<td>11010</td>
<td>denbex</td>
<td>X</td>
<td>Z23</td>
<td></td>
</tr>
<tr>
<td>11011</td>
<td>dix</td>
<td>X</td>
<td>Z23</td>
<td></td>
</tr>
<tr>
<td>11100</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11101</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11110</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11111</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1164  Power ISA™ Appendices
Table 22: EXT59: Extended Opcode Map for Primary Opcode 59 (opcode bits 21:30) (Sheet 2 of 4)
Table 22: EXT59: Extended Opcode Map for Primary Opcode 59 (opcode bits 21:30) (Sheet 3 of 4)

<table>
<thead>
<tr>
<th>Opcode Bits 21:30</th>
<th>fdivs()</th>
<th>fsubs()</th>
<th>fadds()</th>
<th>fsqrts()</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>fdivs()</td>
<td>fsubs()</td>
<td>fadds()</td>
<td>fsqrts()</td>
</tr>
<tr>
<td>00001</td>
<td>fdivs()</td>
<td>fsubs()</td>
<td>fadds()</td>
<td>fsqrts()</td>
</tr>
<tr>
<td>00010</td>
<td>fdivs()</td>
<td>fsubs()</td>
<td>fadds()</td>
<td>fsqrts()</td>
</tr>
<tr>
<td>00011</td>
<td>fdivs()</td>
<td>fsubs()</td>
<td>fadds()</td>
<td>fsqrts()</td>
</tr>
<tr>
<td>00100</td>
<td>fdivs()</td>
<td>fsubs()</td>
<td>fadds()</td>
<td>fsqrts()</td>
</tr>
<tr>
<td>00101</td>
<td>fdivs()</td>
<td>fsubs()</td>
<td>fadds()</td>
<td>fsqrts()</td>
</tr>
<tr>
<td>00110</td>
<td>fdivs()</td>
<td>fsubs()</td>
<td>fadds()</td>
<td>fsqrts()</td>
</tr>
<tr>
<td>00111</td>
<td>fdivs()</td>
<td>fsubs()</td>
<td>fadds()</td>
<td>fsqrts()</td>
</tr>
<tr>
<td>01000</td>
<td>fdivs()</td>
<td>fsubs()</td>
<td>fadds()</td>
<td>fsqrts()</td>
</tr>
<tr>
<td>01001</td>
<td>fdivs()</td>
<td>fsubs()</td>
<td>fadds()</td>
<td>fsqrts()</td>
</tr>
<tr>
<td>01010</td>
<td>fdivs()</td>
<td>fsubs()</td>
<td>fadds()</td>
<td>fsqrts()</td>
</tr>
<tr>
<td>01011</td>
<td>fdivs()</td>
<td>fsubs()</td>
<td>fadds()</td>
<td>fsqrts()</td>
</tr>
<tr>
<td>01100</td>
<td>fdivs()</td>
<td>fsubs()</td>
<td>fadds()</td>
<td>fsqrts()</td>
</tr>
<tr>
<td>01101</td>
<td>fdivs()</td>
<td>fsubs()</td>
<td>fadds()</td>
<td>fsqrts()</td>
</tr>
<tr>
<td>01110</td>
<td>fdivs()</td>
<td>fsubs()</td>
<td>fadds()</td>
<td>fsqrts()</td>
</tr>
<tr>
<td>01111</td>
<td>fdivs()</td>
<td>fsubs()</td>
<td>fadds()</td>
<td>fsqrts()</td>
</tr>
<tr>
<td>10000</td>
<td>fdivs()</td>
<td>fsubs()</td>
<td>fadds()</td>
<td>fsqrts()</td>
</tr>
<tr>
<td>10001</td>
<td>fdivs()</td>
<td>fsubs()</td>
<td>fadds()</td>
<td>fsqrts()</td>
</tr>
<tr>
<td>10010</td>
<td>fdivs()</td>
<td>fsubs()</td>
<td>fadds()</td>
<td>fsqrts()</td>
</tr>
<tr>
<td>10011</td>
<td>fdivs()</td>
<td>fsubs()</td>
<td>fadds()</td>
<td>fsqrts()</td>
</tr>
<tr>
<td>10100</td>
<td>fdivs()</td>
<td>fsubs()</td>
<td>fadds()</td>
<td>fsqrts()</td>
</tr>
<tr>
<td>10101</td>
<td>fdivs()</td>
<td>fsubs()</td>
<td>fadds()</td>
<td>fsqrts()</td>
</tr>
<tr>
<td>10110</td>
<td>fdivs()</td>
<td>fsubs()</td>
<td>fadds()</td>
<td>fsqrts()</td>
</tr>
<tr>
<td>10111</td>
<td>fdivs()</td>
<td>fsubs()</td>
<td>fadds()</td>
<td>fsqrts()</td>
</tr>
<tr>
<td>11000</td>
<td>fdivs()</td>
<td>fsubs()</td>
<td>fadds()</td>
<td>fsqrts()</td>
</tr>
<tr>
<td>11001</td>
<td>fdivs()</td>
<td>fsubs()</td>
<td>fadds()</td>
<td>fsqrts()</td>
</tr>
<tr>
<td>11010</td>
<td>fdivs()</td>
<td>fsubs()</td>
<td>fadds()</td>
<td>fsqrts()</td>
</tr>
<tr>
<td>11011</td>
<td>fdivs()</td>
<td>fsubs()</td>
<td>fadds()</td>
<td>fsqrts()</td>
</tr>
<tr>
<td>11100</td>
<td>fdivs()</td>
<td>fsubs()</td>
<td>fadds()</td>
<td>fsqrts()</td>
</tr>
<tr>
<td>11101</td>
<td>fdivs()</td>
<td>fsubs()</td>
<td>fadds()</td>
<td>fsqrts()</td>
</tr>
<tr>
<td>11110</td>
<td>fdivs()</td>
<td>fsubs()</td>
<td>fadds()</td>
<td>fsqrts()</td>
</tr>
<tr>
<td>11111</td>
<td>fdivs()</td>
<td>fsubs()</td>
<td>fadds()</td>
<td>fsqrts()</td>
</tr>
</tbody>
</table>
Table 22: EXT59: Extended Opcode Map for Primary Opcode 59 (opcode bits 21:30) (Sheet 4 of 4)
Table 23: EXT60: Extended Opcode Map for Primary Opcode 60 (opcode 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>0000</td>
<td>xsaddsp</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0001</td>
<td>xsaddsp</td>
<td>00000</td>
<td>00001</td>
<td>00010</td>
<td>00011</td>
<td>00100</td>
<td>00101</td>
<td>00110</td>
</tr>
<tr>
<td>0010</td>
<td>xsmulp</td>
<td>00000</td>
<td>00001</td>
<td>00010</td>
<td>00011</td>
<td>00100</td>
<td>00101</td>
<td>00110</td>
</tr>
<tr>
<td>0011</td>
<td>xdivsp</td>
<td>00000</td>
<td>00001</td>
<td>00010</td>
<td>00011</td>
<td>00100</td>
<td>00101</td>
<td>00110</td>
</tr>
<tr>
<td>0100</td>
<td>xaddsp</td>
<td>00000</td>
<td>00001</td>
<td>00010</td>
<td>00011</td>
<td>00100</td>
<td>00101</td>
<td>00110</td>
</tr>
<tr>
<td>0101</td>
<td>xsmulp</td>
<td>00000</td>
<td>00001</td>
<td>00010</td>
<td>00011</td>
<td>00100</td>
<td>00101</td>
<td>00110</td>
</tr>
<tr>
<td>0110</td>
<td>xdivsp</td>
<td>00000</td>
<td>00001</td>
<td>00010</td>
<td>00011</td>
<td>00100</td>
<td>00101</td>
<td>00110</td>
</tr>
<tr>
<td>0111</td>
<td>xsmulp</td>
<td>00000</td>
<td>00001</td>
<td>00010</td>
<td>00011</td>
<td>00100</td>
<td>00101</td>
<td>00110</td>
</tr>
<tr>
<td>1000</td>
<td>xsmaxdp</td>
<td>00000</td>
<td>00001</td>
<td>00010</td>
<td>00011</td>
<td>00100</td>
<td>00101</td>
<td>00110</td>
</tr>
<tr>
<td>1001</td>
<td>xsmindp</td>
<td>00000</td>
<td>00001</td>
<td>00010</td>
<td>00011</td>
<td>00100</td>
<td>00101</td>
<td>00110</td>
</tr>
<tr>
<td>1010</td>
<td>xsmaxdp</td>
<td>00000</td>
<td>00001</td>
<td>00010</td>
<td>00011</td>
<td>00100</td>
<td>00101</td>
<td>00110</td>
</tr>
<tr>
<td>1011</td>
<td>xsmindp</td>
<td>00000</td>
<td>00001</td>
<td>00010</td>
<td>00011</td>
<td>00100</td>
<td>00101</td>
<td>00110</td>
</tr>
<tr>
<td>1100</td>
<td>xscpsgndp</td>
<td>00000</td>
<td>00001</td>
<td>00010</td>
<td>00011</td>
<td>00100</td>
<td>00101</td>
<td>00110</td>
</tr>
<tr>
<td>1101</td>
<td>xscpsgndp</td>
<td>00000</td>
<td>00001</td>
<td>00010</td>
<td>00011</td>
<td>00100</td>
<td>00101</td>
<td>00110</td>
</tr>
<tr>
<td>1110</td>
<td>xscpsgndp</td>
<td>00000</td>
<td>00001</td>
<td>00010</td>
<td>00011</td>
<td>00100</td>
<td>00101</td>
<td>00110</td>
</tr>
<tr>
<td>1111</td>
<td>xscpsgndp</td>
<td>00000</td>
<td>00001</td>
<td>00010</td>
<td>00011</td>
<td>00100</td>
<td>00101</td>
<td>00110</td>
</tr>
</tbody>
</table>
Table 23: EXT60: Extended Opcode Map for Primary Opcode 60 (opcode bits 21:30) (Sheet 2 of 4)
Table 23: EXT60: Extended Opcode Map for Primary Opcode 60 (opcode bits 21:30) (Sheet 3 of 4)

<table>
<thead>
<tr>
<th>Opcode</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>0000</td>
</tr>
<tr>
<td>0001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0001</td>
</tr>
<tr>
<td>0010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0010</td>
</tr>
<tr>
<td>0011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0011</td>
</tr>
<tr>
<td>0100</td>
<td>xscvdpxws</td>
<td>xscvdpxws</td>
<td>xscvdpxws</td>
<td>xscvdpxws</td>
<td>xscvdpxws</td>
<td>xscvdpxws</td>
<td>xscvdpxws</td>
<td>xscvdpxws</td>
</tr>
<tr>
<td></td>
<td>0.26</td>
<td>0.26</td>
<td>0.26</td>
<td>0.26</td>
<td>0.26</td>
<td>0.26</td>
<td>0.26</td>
<td>0.26</td>
</tr>
<tr>
<td>0101</td>
<td>xscvdpsxws</td>
<td>xscvdpsxws</td>
<td>xscvdpsxws</td>
<td>xscvdpsxws</td>
<td>xscvdpsxws</td>
<td>xscvdpsxws</td>
<td>xscvdpsxws</td>
<td>xscvdpsxws</td>
</tr>
<tr>
<td></td>
<td>0.26</td>
<td>0.26</td>
<td>0.26</td>
<td>0.26</td>
<td>0.26</td>
<td>0.26</td>
<td>0.26</td>
<td>0.26</td>
</tr>
<tr>
<td>0110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0110</td>
</tr>
<tr>
<td>0111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0111</td>
</tr>
<tr>
<td>1000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1000</td>
</tr>
<tr>
<td>1001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1001</td>
</tr>
<tr>
<td>1010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1010</td>
</tr>
<tr>
<td>1011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1011</td>
</tr>
<tr>
<td>1100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1100</td>
</tr>
<tr>
<td>1101</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1101</td>
</tr>
<tr>
<td>1110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1110</td>
</tr>
<tr>
<td>1111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1111</td>
</tr>
</tbody>
</table>
Table 23: EXT60: Extended Opcode Map for Primary Opcode 60 (opcode 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></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>

xxsel
XX4
Table 24: XPND60-1: Extended Opcode Map for PO=60 XO=0b01011_01000 (opcode 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 25: XPND60-2: Extended Opcode Map for PO=60 XO=0b10101_1011- (opcode 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 26: XPND60-3: Extended Opcode Map for PO=60 XO=0b11101_1011- (opcode 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 27: EXT63: Extended Opcode Map for Primary Opcode 63 (opcode bits 21:30) (Sheet 1 of 4)

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>fcmpu</td>
<td>Compare Float Registers</td>
</tr>
<tr>
<td>00001</td>
<td>fcmpeq</td>
<td>Compare Float Registers</td>
</tr>
<tr>
<td>00010</td>
<td>mcrfs</td>
<td>Compare Register Flags</td>
</tr>
<tr>
<td>00011</td>
<td>fcmpeq</td>
<td>Compare Float Registers</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01000</td>
<td>fdiv</td>
<td>Divide Float</td>
</tr>
<tr>
<td>01001</td>
<td>fdiv</td>
<td>Divide Float</td>
</tr>
<tr>
<td>01010</td>
<td>dtsqrt</td>
<td>Square Root Double</td>
</tr>
<tr>
<td>01011</td>
<td>dtsqrt</td>
<td>Square Root Double</td>
</tr>
<tr>
<td>01100</td>
<td>dtstdcq</td>
<td>Standard Double</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>dtsubq</td>
<td>Subtract Double</td>
</tr>
<tr>
<td>10001</td>
<td>dtsubq</td>
<td>Subtract Double</td>
</tr>
<tr>
<td>10010</td>
<td>dtsubq</td>
<td>Subtract Double</td>
</tr>
<tr>
<td>10011</td>
<td>dtsubq</td>
<td>Subtract Double</td>
</tr>
<tr>
<td>10100</td>
<td>dtsubq</td>
<td>Subtract Double</td>
</tr>
<tr>
<td>10101</td>
<td>dtsubq</td>
<td>Subtract Double</td>
</tr>
<tr>
<td>10110</td>
<td>dtsubq</td>
<td>Subtract Double</td>
</tr>
<tr>
<td>10111</td>
<td>dtsubq</td>
<td>Subtract Double</td>
</tr>
<tr>
<td>11000</td>
<td>dtsubq</td>
<td>Subtract Double</td>
</tr>
<tr>
<td>11001</td>
<td>dtsubq</td>
<td>Subtract Double</td>
</tr>
<tr>
<td>11010</td>
<td>dtsubq</td>
<td>Subtract Double</td>
</tr>
<tr>
<td>11011</td>
<td>dtsubq</td>
<td>Subtract Double</td>
</tr>
<tr>
<td>11100</td>
<td>dtsubq</td>
<td>Subtract Double</td>
</tr>
<tr>
<td>11101</td>
<td>dtsubq</td>
<td>Subtract Double</td>
</tr>
<tr>
<td>11110</td>
<td>dtsubq</td>
<td>Subtract Double</td>
</tr>
<tr>
<td>11111</td>
<td>dtsubq</td>
<td>Subtract Double</td>
</tr>
</tbody>
</table>

**Legend:**
- **X:** Not supported
- **I:** Instruction
- **FL:** Floating Point
- **XFL:** Floating Point (expanded)
- **P1:** Special function not supported
- **EXP:** Expanded operation
- **XPND63:** Expanded operation

---

**Version 3.0 B**

**Appendix C. Opcode Maps** 1173
### Table 27: EXT63: Extended Opcode Map for Primary Opcode 63 (opcode bits 21:30) (Sheet 2 of 4)

<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>fcpsgn[,]</td>
<td>x</td>
<td>frsp[,]</td>
<td>x</td>
<td>fctiw[,]</td>
<td>x</td>
<td>fctiwz[,]</td>
<td>x</td>
</tr>
<tr>
<td>0001</td>
<td>fneg[,]</td>
<td>x</td>
<td>frsp[,]</td>
<td>x</td>
<td>fctiw[,]</td>
<td>x</td>
<td>fctiwz[,]</td>
<td>x</td>
</tr>
<tr>
<td>0010</td>
<td>frm[,]</td>
<td>x</td>
<td>frsp[,]</td>
<td>x</td>
<td>fctiw[,]</td>
<td>x</td>
<td>fctiwz[,]</td>
<td>x</td>
</tr>
<tr>
<td>0011</td>
<td>frsp[,]</td>
<td>x</td>
<td>frsp[,]</td>
<td>x</td>
<td>fctiw[,]</td>
<td>x</td>
<td>fctiwz[,]</td>
<td>x</td>
</tr>
<tr>
<td>0100</td>
<td>fnsb[,]</td>
<td>x</td>
<td>fctiw[,]</td>
<td>x</td>
<td>fctiwz[,]</td>
<td>x</td>
<td>0100</td>
<td></td>
</tr>
<tr>
<td>0101</td>
<td>fnsb[,]</td>
<td>x</td>
<td>fctiw[,]</td>
<td>x</td>
<td>fctiwz[,]</td>
<td>x</td>
<td>0101</td>
<td></td>
</tr>
<tr>
<td>0110</td>
<td>fnsb[,]</td>
<td>x</td>
<td>fctiw[,]</td>
<td>x</td>
<td>fctiwz[,]</td>
<td>x</td>
<td>0110</td>
<td></td>
</tr>
<tr>
<td>0111</td>
<td>fnsb[,]</td>
<td>x</td>
<td>fctiw[,]</td>
<td>x</td>
<td>fctiwz[,]</td>
<td>x</td>
<td>0111</td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>fabs[,]</td>
<td>x</td>
<td>fctid[,]</td>
<td>x</td>
<td>fctidz[,]</td>
<td>x</td>
<td>1000</td>
<td></td>
</tr>
<tr>
<td>1001</td>
<td>fabs[,]</td>
<td>x</td>
<td>fctid[,]</td>
<td>x</td>
<td>fctidz[,]</td>
<td>x</td>
<td>1001</td>
<td></td>
</tr>
<tr>
<td>1010</td>
<td>fabs[,]</td>
<td>x</td>
<td>fctid[,]</td>
<td>x</td>
<td>fctidz[,]</td>
<td>x</td>
<td>1010</td>
<td></td>
</tr>
<tr>
<td>1011</td>
<td>fabs[,]</td>
<td>x</td>
<td>fctid[,]</td>
<td>x</td>
<td>fctidz[,]</td>
<td>x</td>
<td>1011</td>
<td></td>
</tr>
<tr>
<td>1100</td>
<td>fabs[,]</td>
<td>x</td>
<td>fctid[,]</td>
<td>x</td>
<td>fctidz[,]</td>
<td>x</td>
<td>1100</td>
<td></td>
</tr>
<tr>
<td>1101</td>
<td>fabs[,]</td>
<td>x</td>
<td>fctid[,]</td>
<td>x</td>
<td>fctidz[,]</td>
<td>x</td>
<td>1101</td>
<td></td>
</tr>
<tr>
<td>1110</td>
<td>fabs[,]</td>
<td>x</td>
<td>fctid[,]</td>
<td>x</td>
<td>fctidz[,]</td>
<td>x</td>
<td>1110</td>
<td></td>
</tr>
<tr>
<td>1111</td>
<td>fabs[,]</td>
<td>x</td>
<td>fctid[,]</td>
<td>x</td>
<td>fctidz[,]</td>
<td>x</td>
<td>1111</td>
<td></td>
</tr>
</tbody>
</table>
Table 27: EXT63: Extended Opcode Map for Primary Opcode 63 (opcode bits 21:30) (Sheet 3 of 4)
Table 27: EXT63: Extended Opcode Map for Primary Opcode 63 (opcode bits 21:30) (Sheet 4 of 4)
### Appendix C. Opcode Maps

#### Version 3.0 B

Table 28: XPND63-1: Extended Opcode Map for PO=63 XO=0b10010_00111 (opcode bits 11:15)

<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>00</td>
<td>mfs[.]</td>
<td>0.0B</td>
<td>mffsce</td>
<td>x</td>
<td>0.0B</td>
<td>mffscdrn</td>
<td>v3.0B</td>
</tr>
<tr>
<td>01</td>
<td>0.0B</td>
<td>mffsce</td>
<td>x</td>
<td>0.0B</td>
<td>mffscdrn</td>
<td>v3.0B</td>
<td>X</td>
</tr>
<tr>
<td>10</td>
<td>0.0B</td>
<td>mffsce</td>
<td>x</td>
<td>0.0B</td>
<td>mffscdrn</td>
<td>v3.0B</td>
<td>X</td>
</tr>
<tr>
<td>11</td>
<td>0.0B</td>
<td>mffsce</td>
<td>x</td>
<td>0.0B</td>
<td>mffscdrn</td>
<td>v3.0B</td>
<td>X</td>
</tr>
</tbody>
</table>

Table 29: XPND63-2: Extended Opcode Map for PO=63 XO=0b11001_00100 (opcode bits 11:15)

<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>00</td>
<td>xsabsqp</td>
<td>x</td>
<td>xsxsqpp</td>
<td>x</td>
<td>0.0</td>
<td>xsxsqpp</td>
<td>x</td>
</tr>
<tr>
<td>01</td>
<td>xsabsqp</td>
<td>x</td>
<td>xsxsqpp</td>
<td>x</td>
<td>0.0</td>
<td>xsxsqpp</td>
<td>x</td>
</tr>
<tr>
<td>10</td>
<td>xsabsqp</td>
<td>x</td>
<td>xsxsqpp</td>
<td>x</td>
<td>0.0</td>
<td>xsxsqpp</td>
<td>x</td>
</tr>
<tr>
<td>11</td>
<td>xsabsqp</td>
<td>x</td>
<td>xsxsqpp</td>
<td>x</td>
<td>0.0</td>
<td>xsxsqpp</td>
<td>x</td>
</tr>
</tbody>
</table>

Table 30: XPND63-3: Extended Opcode Map for PO=63 XO=0b11010_00100 (opcode bits 11:15)

<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>00</td>
<td>xscvqpuwz</td>
<td>x</td>
<td>xscvqdp [o]</td>
<td>x</td>
<td>0.0</td>
<td>xscvqdp [o]</td>
<td>x</td>
</tr>
<tr>
<td>01</td>
<td>xscvqpswz</td>
<td>x</td>
<td>xscvqdp [o]</td>
<td>x</td>
<td>0.0</td>
<td>xscvqdp [o]</td>
<td>x</td>
</tr>
<tr>
<td>10</td>
<td>xscvqpswz</td>
<td>x</td>
<td>xscvqdp [o]</td>
<td>x</td>
<td>0.0</td>
<td>xscvqdp [o]</td>
<td>x</td>
</tr>
<tr>
<td>11</td>
<td>xscvqpswz</td>
<td>x</td>
<td>xscvqdp [o]</td>
<td>x</td>
<td>0.0</td>
<td>xscvqdp [o]</td>
<td>x</td>
</tr>
</tbody>
</table>
Appendix D. Power ISA Instruction Set Sorted by Opcode

This appendix lists all the instructions in the Power ISA, sorted by primary opcode, then by extended opcode bits 26:31 (if any), then by opcode bits 21:25 (if any), then by expanded opcode bits 11:15 (if any).

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>Book</th>
<th>Page</th>
<th>Mnemonic</th>
<th>Version</th>
<th>Privilege</th>
<th>Mode Dep</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>
</tr>
</tbody>
</table>

Figure 88. Power ISA AS Instruction Set Sorted by Opcode (Sheet 1 of 18)
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>Book</th>
<th>Page</th>
<th>Mnemonic</th>
<th>Version</th>
<th>Privilege</th>
<th>Mode Dep</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>001000</td>
<td>VX</td>
<td>354</td>
<td>bcdszq</td>
<td>v3.0</td>
<td>Decimal Convert To Signed Quadword &amp; record</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001100</td>
<td>VX</td>
<td>354</td>
<td>bcdszq</td>
<td>v3.0</td>
<td>Decimal Convert From Signed Quadword &amp; record</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001100</td>
<td>VX</td>
<td>353</td>
<td>bcdsz</td>
<td>v3.0</td>
<td>Decimal Convert To Zoned &amp; record</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001100</td>
<td>VX</td>
<td>352</td>
<td>bcdsh</td>
<td>v3.0</td>
<td>Decimal Convert To National &amp; record</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001100</td>
<td>VX</td>
<td>351</td>
<td>bcdszl</td>
<td>v3.0</td>
<td>Decimal Convert From Zoned &amp; record</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001100</td>
<td>VX</td>
<td>350</td>
<td>bcdch</td>
<td>v3.0</td>
<td>Decimal Convert From National &amp; record</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011000</td>
<td>VX</td>
<td>349</td>
<td>bcds</td>
<td>v3.0</td>
<td>Decimal Shift &amp; Round &amp; record</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000000</td>
<td>VX</td>
<td>298</td>
<td>vmaxub</td>
<td>v2.0</td>
<td>Vector Maximum Unsigned Byte</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000000</td>
<td>VX</td>
<td>290</td>
<td>vmaxuh</td>
<td>v2.0</td>
<td>Vector Maximum Unsigned Halfword</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000000</td>
<td>VX</td>
<td>290</td>
<td>vmaxuw</td>
<td>v2.0</td>
<td>Vector Maximum Unsigned Word</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000000</td>
<td>VX</td>
<td>290</td>
<td>vmaxud</td>
<td>v2.07</td>
<td>Vector Maximum Unsigned Doubleword</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000000</td>
<td>VX</td>
<td>309</td>
<td>vmaxsb</td>
<td>v2.0</td>
<td>Vector Maximum Signed Byte</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000000</td>
<td>VX</td>
<td>309</td>
<td>vmaxsh</td>
<td>v2.0</td>
<td>Vector Maximum Signed Halfword</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000000</td>
<td>VX</td>
<td>309</td>
<td>vmaxsw</td>
<td>v2.0</td>
<td>Vector Maximum Signed Word</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000000</td>
<td>VX</td>
<td>309</td>
<td>vmaxsd</td>
<td>v2.0</td>
<td>Vector Maximum Signed Doubleword</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000000</td>
<td>VX</td>
<td>301</td>
<td>vminub</td>
<td>v2.0</td>
<td>Vector Minimum Unsigned Byte</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000000</td>
<td>VX</td>
<td>301</td>
<td>vminuh</td>
<td>v2.0</td>
<td>Vector Minimum Unsigned Halfword</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000000</td>
<td>VX</td>
<td>301</td>
<td>vminuw</td>
<td>v2.0</td>
<td>Vector Minimum Unsigned Word</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000000</td>
<td>VX</td>
<td>301</td>
<td>vminud</td>
<td>v2.0</td>
<td>Vector Minimum Signed Doubleword</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000000</td>
<td>VX</td>
<td>301</td>
<td>vminsh</td>
<td>v2.0</td>
<td>Vector Minimum Signed Halfword</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000000</td>
<td>VX</td>
<td>301</td>
<td>vminsw</td>
<td>v2.0</td>
<td>Vector Minimum Signed Word</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000000</td>
<td>VX</td>
<td>301</td>
<td>vminsds</td>
<td>v2.07</td>
<td>Vector Minimum Signed Doubleword</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000000</td>
<td>VX</td>
<td>296</td>
<td>vavgub</td>
<td>v2.0</td>
<td>Vector Average Unsigned Byte</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000000</td>
<td>VX</td>
<td>296</td>
<td>vavguh</td>
<td>v2.0</td>
<td>Vector Average Unsigned Halfword</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000000</td>
<td>VX</td>
<td>296</td>
<td>vavguw</td>
<td>v2.0</td>
<td>Vector Average Unsigned Word</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000000</td>
<td>VX</td>
<td>295</td>
<td>vavgub</td>
<td>v2.0</td>
<td>Vector Average Signed Byte</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000000</td>
<td>VX</td>
<td>295</td>
<td>vavgsh</td>
<td>v2.0</td>
<td>Vector Average Signed Halfword</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000000</td>
<td>VX</td>
<td>295</td>
<td>vavgsw</td>
<td>v2.0</td>
<td>Vector Average Signed Word</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000000</td>
<td>VX</td>
<td>342</td>
<td>vcdlsbb</td>
<td>v3.0</td>
<td>Vector Count Leading Zero Least-Significant Bits Byte</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000000</td>
<td>VX</td>
<td>342</td>
<td>vcdlbsbb</td>
<td>v3.0</td>
<td>Vector Count Trailing Zero Least-Significant Bits Byte</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td>VX</td>
<td>293</td>
<td>vnegw</td>
<td>v3.0</td>
<td>Vector Negate Word</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td>VX</td>
<td>293</td>
<td>vnegd</td>
<td>v3.0</td>
<td>Vector Negate Doubleword</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td>VX</td>
<td>293</td>
<td>vnegl</td>
<td>v3.0</td>
<td>Vector Negate Longword</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td>VX</td>
<td>314</td>
<td>vprtybw</td>
<td>v3.0</td>
<td>Vector Parity Byte Word</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td>VX</td>
<td>314</td>
<td>vprtybd</td>
<td>v3.0</td>
<td>Vector Parity Byte Doubleword</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td>VX</td>
<td>314</td>
<td>vprtybq</td>
<td>v3.0</td>
<td>Vector Parity Byte Quadword</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001000</td>
<td>VX</td>
<td>294</td>
<td>vextb2w</td>
<td>v3.0</td>
<td>Vector Extend Sign Byte to Word</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001000</td>
<td>VX</td>
<td>294</td>
<td>vextb2h</td>
<td>v3.0</td>
<td>Vector Extend Sign Halfword to Word</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001000</td>
<td>VX</td>
<td>294</td>
<td>vextb2d</td>
<td>v3.0</td>
<td>Vector Extend Sign Doubleword to Word</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001000</td>
<td>VX</td>
<td>294</td>
<td>vextb2cl</td>
<td>v3.0</td>
<td>Vector Extend Sign Longword to Word</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001000</td>
<td>VX</td>
<td>294</td>
<td>vextb2check</td>
<td>v3.0</td>
<td>Vector Extend Sign Check Word to Word</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001000</td>
<td>VX</td>
<td>294</td>
<td>vextb2check</td>
<td>v3.0</td>
<td>Vector Extend Sign Check Longword to Word</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001000</td>
<td>VX</td>
<td>341</td>
<td>vcztb</td>
<td>v3.0</td>
<td>Vector Count Trailing Zeros Byte</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001000</td>
<td>VX</td>
<td>341</td>
<td>vcztw</td>
<td>v3.0</td>
<td>Vector Count Trailing Zeros Word</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001000</td>
<td>VX</td>
<td>341</td>
<td>vcztwh</td>
<td>v3.0</td>
<td>Vector Count Trailing Zeros Halfword</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001000</td>
<td>VX</td>
<td>341</td>
<td>vcztwd</td>
<td>v3.0</td>
<td>Vector Count Trailing Zeros Doubleword</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001000</td>
<td>VX</td>
<td>335</td>
<td>vshasigmaw</td>
<td>v2.0</td>
<td>Vector SHA-256 Sigma Word</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001000</td>
<td>VX</td>
<td>335</td>
<td>vshasigmad</td>
<td>v2.0</td>
<td>Vector SHA-512 Sigma Doubleword</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001000</td>
<td>VX</td>
<td>340</td>
<td>vcdzb</td>
<td>v2.0</td>
<td>Vector Count Leading Zeros Byte</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001000</td>
<td>VX</td>
<td>340</td>
<td>vcdzh</td>
<td>v2.0</td>
<td>Vector Count Leading Zeros Halfword</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001000</td>
<td>VX</td>
<td>340</td>
<td>vcdzw</td>
<td>v2.0</td>
<td>Vector Count Leading Zeros Word</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001000</td>
<td>VX</td>
<td>340</td>
<td>vcdzd</td>
<td>v2.0</td>
<td>Vector Count Leading Zeros Doubleword</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001000</td>
<td>VX</td>
<td>297</td>
<td>vabsdb</td>
<td>v3.0</td>
<td>Vector Absolute Difference Signed Byte</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001000</td>
<td>VX</td>
<td>297</td>
<td>vabsdh</td>
<td>v3.0</td>
<td>Vector Absolute Difference Signed Halfword</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001000</td>
<td>VX</td>
<td>297</td>
<td>vabsdw</td>
<td>v3.0</td>
<td>Vector Absolute Difference Signed Word</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001000</td>
<td>VX</td>
<td>297</td>
<td>vabsdwb</td>
<td>v3.0</td>
<td>Vector Absolute Difference Unsigned Word</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001000</td>
<td>VX</td>
<td>345</td>
<td>vpopcntb</td>
<td>v2.0</td>
<td>Vector Population Count Byte</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 88. Power ISA AS Instruction Set Sorted by Opcode (Sheet 2 of 18)
<table>
<thead>
<tr>
<th>Instruction1</th>
<th>Format</th>
<th>Book</th>
<th>Page</th>
<th>Mnemonic</th>
<th>Version2</th>
<th>Privilege3</th>
<th>Mode4</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>000100</td>
<td>I</td>
<td>345</td>
<td>vyppcnth</td>
<td>v2.07</td>
<td>Vector Population Count Halfword</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td>I</td>
<td>345</td>
<td>vyppcnw</td>
<td>v2.07</td>
<td>Vector Population Count Word</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td>I</td>
<td>345</td>
<td>vyppcond</td>
<td>v2.07</td>
<td>Vector Population Count Doubleword</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td>I</td>
<td>315</td>
<td>vrib</td>
<td>v2.03</td>
<td>Vector Rotate Left Byte</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td>I</td>
<td>315</td>
<td>vribL</td>
<td>v2.03</td>
<td>Vector Rotate Left Halfword</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td>I</td>
<td>315</td>
<td>vribW</td>
<td>v2.03</td>
<td>Vector Rotate Left Word</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td>I</td>
<td>315</td>
<td>vribD</td>
<td>v2.07</td>
<td>Vector Rotate Left Doubleword</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td>I</td>
<td>316</td>
<td>vish</td>
<td>v2.03</td>
<td>Vector Shift Left Byte</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td>I</td>
<td>316</td>
<td>vishL</td>
<td>v2.03</td>
<td>Vector Shift Left Halfword</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td>I</td>
<td>264</td>
<td>vsl</td>
<td>v2.03</td>
<td>Vector Shift Left</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td>I</td>
<td>317</td>
<td>vslr</td>
<td>v2.03</td>
<td>Vector Shift Right Byte</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td>I</td>
<td>317</td>
<td>vslrH</td>
<td>v2.03</td>
<td>Vector Shift Right Halfword</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td>I</td>
<td>317</td>
<td>vslrW</td>
<td>v2.03</td>
<td>Vector Shift Right Word</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td>I</td>
<td>264</td>
<td>vslrD</td>
<td>v2.03</td>
<td>Vector Shift Right Doubleword</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td>I</td>
<td>318</td>
<td>vrsab</td>
<td>v2.03</td>
<td>Vector Shift Right Algebraic Byte</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td>I</td>
<td>318</td>
<td>vrsah</td>
<td>v2.03</td>
<td>Vector Shift Right Algebraic Halfword</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td>I</td>
<td>318</td>
<td>vrsaw</td>
<td>v2.03</td>
<td>Vector Shift Right Algebraic Word</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td>I</td>
<td>318</td>
<td>vrsad</td>
<td>v2.07</td>
<td>Vector Shift Right Algebraic Doubleword</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td>I</td>
<td>312</td>
<td>vand</td>
<td>v2.03</td>
<td>Vector Logical AND</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td>I</td>
<td>312</td>
<td>vandc</td>
<td>v2.03</td>
<td>Vector Logical AND with Complement</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td>I</td>
<td>313</td>
<td>vor</td>
<td>v2.03</td>
<td>Vector Logical OR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td>I</td>
<td>313</td>
<td>vorc</td>
<td>v2.03</td>
<td>Vector Logical XOR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td>I</td>
<td>315</td>
<td>vorc</td>
<td>v2.03</td>
<td>Vector Logical NOR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td>I</td>
<td>315</td>
<td>vors</td>
<td>v2.03</td>
<td>Vector OR with Complement</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td>I</td>
<td>312</td>
<td>vnand</td>
<td>v2.07</td>
<td>Vector NAND</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td>I</td>
<td>316</td>
<td>vsl</td>
<td>v2.07</td>
<td>Vector Shift Left Doubleword</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td>I</td>
<td>362</td>
<td>mfvscr</td>
<td>v2.03</td>
<td>Move From VSCR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td>I</td>
<td>362</td>
<td>mfvscw</td>
<td>v2.03</td>
<td>Move To VSCR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td>I</td>
<td>312</td>
<td>veqv</td>
<td>v2.07</td>
<td>Vector Equivalence</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td>I</td>
<td>317</td>
<td>vadd</td>
<td>v2.07</td>
<td>Vector Shift Right Doubleword</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td>I</td>
<td>316</td>
<td>vsv</td>
<td>v3.0</td>
<td>Vector Shift Right Variable</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td>I</td>
<td>325</td>
<td>vsvl</td>
<td>v3.0</td>
<td>Vector Shift Left Variable</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td>I</td>
<td>320</td>
<td>vslr</td>
<td>v3.0</td>
<td>Vector Rotate Left Word then Mask Insert</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td>I</td>
<td>320</td>
<td>vslrh</td>
<td>v3.0</td>
<td>Vector Rotate Left Doubleword then Mask Insert</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td>I</td>
<td>320</td>
<td>vslrw</td>
<td>v3.0</td>
<td>Vector Rotate Left Word then AND with Mask</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td>I</td>
<td>320</td>
<td>vslrd</td>
<td>v3.0</td>
<td>Vector Rotate Left Doubleword then AND with Mask</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td>I</td>
<td>303</td>
<td>vcmpueq5</td>
<td>v2.03</td>
<td>Vector Compare Equal To Signed Byte</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td>I</td>
<td>303</td>
<td>vcmpueq6</td>
<td>v2.03</td>
<td>Vector Compare Equal To Signed Word</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td>I</td>
<td>303</td>
<td>vcmpueq7</td>
<td>v2.03</td>
<td>Vector Compare Equal To Signed Halfword</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td>I</td>
<td>303</td>
<td>vcmpueq8</td>
<td>v2.03</td>
<td>Vector Compare Equal To Signed Doubleword</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td>I</td>
<td>303</td>
<td>vcmpueq9</td>
<td>v2.03</td>
<td>Vector Compare Equal To Signed Tripleword</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td>I</td>
<td>303</td>
<td>vcmpueq10</td>
<td>v2.03</td>
<td>Vector Compare Equal To Signed Quadword</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td>I</td>
<td>303</td>
<td>vcmpueq11</td>
<td>v2.03</td>
<td>Vector Compare Equal To Signed Pentadword</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td>I</td>
<td>303</td>
<td>vcmpueq12</td>
<td>v2.03</td>
<td>Vector Compare Equal To Signed Hexadword</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td>I</td>
<td>303</td>
<td>vcmpueq13</td>
<td>v2.03</td>
<td>Vector Compare Equal To Signed Octadword</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td>I</td>
<td>303</td>
<td>vcmpueq14</td>
<td>v2.03</td>
<td>Vector Compare Equal To Signed Hexadword</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td>I</td>
<td>303</td>
<td>vcmpueq15</td>
<td>v2.03</td>
<td>Vector Compare Equal To Signed Octadword</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td>I</td>
<td>303</td>
<td>vcmpueq16</td>
<td>v2.03</td>
<td>Vector Compare Equal To Signed Hexadword</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td>I</td>
<td>303</td>
<td>vcmpueq17</td>
<td>v2.03</td>
<td>Vector Compare Equal To Signed Octadword</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td>I</td>
<td>303</td>
<td>vcmpueq18</td>
<td>v2.03</td>
<td>Vector Compare Equal To Signed Hexadword</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td>I</td>
<td>303</td>
<td>vcmpueq19</td>
<td>v2.03</td>
<td>Vector Compare Equal To Signed Octadword</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td>I</td>
<td>303</td>
<td>vcmpueq20</td>
<td>v2.03</td>
<td>Vector Compare Equal To Signed Hexadword</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td>I</td>
<td>303</td>
<td>vcmpueq21</td>
<td>v2.03</td>
<td>Vector Compare Equal To Signed Octadword</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td>I</td>
<td>303</td>
<td>vcmpueq22</td>
<td>v2.03</td>
<td>Vector Compare Equal To Signed Hexadword</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td>I</td>
<td>303</td>
<td>vcmpueq23</td>
<td>v2.03</td>
<td>Vector Compare Equal To Signed Octadword</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td>I</td>
<td>303</td>
<td>vcmpueq24</td>
<td>v2.03</td>
<td>Vector Compare Equal To Signed Hexadword</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td>I</td>
<td>303</td>
<td>vcmpueq25</td>
<td>v2.03</td>
<td>Vector Compare Equal To Signed Octadword</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td>I</td>
<td>303</td>
<td>vcmpueq26</td>
<td>v2.03</td>
<td>Vector Compare Equal To Signed Hexadword</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td>I</td>
<td>303</td>
<td>vcmpueq27</td>
<td>v2.03</td>
<td>Vector Compare Equal To Signed Octadword</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td>I</td>
<td>303</td>
<td>vcmpueq28</td>
<td>v2.03</td>
<td>Vector Compare Equal To Signed Hexadword</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td>I</td>
<td>303</td>
<td>vcmpueq29</td>
<td>v2.03</td>
<td>Vector Compare Equal To Signed Octadword</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td>I</td>
<td>303</td>
<td>vcmpueq30</td>
<td>v2.03</td>
<td>Vector Compare Equal To Signed Hexadword</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td>I</td>
<td>303</td>
<td>vcmpueq31</td>
<td>v2.03</td>
<td>Vector Compare Equal To Signed Octadword</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 88. Power ISA AS Instruction Set Sorted by Opcode (Sheet 3 of 18)

Appendix D. Power ISA Instruction Set Sorted by Opcode  1181
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>Book</th>
<th>Page</th>
<th>Mnemonic</th>
<th>Version</th>
<th>Privilege</th>
<th>Mode Dep</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>00100</td>
<td>VC</td>
<td>0010</td>
<td>0011</td>
<td>vcmpnezw</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>Vector Compare Not Equal or Zero Word</td>
</tr>
<tr>
<td>00100</td>
<td>VC</td>
<td>0010</td>
<td>0011</td>
<td>vcmpltdt</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>Vector Compare Greater Than Unsigned Doubleword</td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>0000</td>
<td>0010</td>
<td>vmultub</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>Vector Multiply Odd Unsigned Byte</td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>0001</td>
<td>0010</td>
<td>vmultuh</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>Vector Multiply Odd Unsigned Halfword</td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>0001</td>
<td>0010</td>
<td>vmultow</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>Vector Multiply Odd Unsigned Word</td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>0010</td>
<td>0010</td>
<td>vmultosb</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>Vector Multiply Odd Signed Byte</td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>0011</td>
<td>0010</td>
<td>vmultosh</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>Vector Multiply Odd Signed Halfword</td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>0011</td>
<td>0010</td>
<td>vmultosw</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>Vector Multiply Odd Signed Word</td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>0100</td>
<td>0010</td>
<td>ymluseb</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>Vector Multiply Even Signed Byte</td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>0101</td>
<td>0010</td>
<td>ymluseh</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>Vector Multiply Even Signed Halfword</td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>0110</td>
<td>0010</td>
<td>ymluesb</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>Vector Multiply Even Signed Word</td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>0111</td>
<td>0010</td>
<td>ymlulesh</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>Vector Multiply Even Signed Halfword</td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>0110</td>
<td>0010</td>
<td>ymlulew</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>Vector Multiply Even Signed Word</td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>0000</td>
<td>0010</td>
<td>vpmumwb</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>Vector Polynomial Multiply-Sum Byte</td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>0001</td>
<td>0010</td>
<td>vpmumw</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>Vector Polynomial Multiply-Sum Halfword</td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>0011</td>
<td>0010</td>
<td>vpmumwb</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>Vector Polynomial Multiply-Sum Wave</td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>0101</td>
<td>0010</td>
<td>vcipher</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>Vector AES Cipher Last</td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>0100</td>
<td>0010</td>
<td>vncipher</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>Vector AES Inverse Cipher Last</td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>0010</td>
<td>0010</td>
<td>vaddflp</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>Vector Add Floating-Point</td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>0001</td>
<td>0010</td>
<td>vsubflp</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>Vector Subtract Floating-Point</td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>0110</td>
<td>0010</td>
<td>vrcelpfl</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>Vector Reciprocal Estimate Floating-Point</td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>0111</td>
<td>0010</td>
<td>vrcshelp</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>Vector Reciprocal Square Root Estimate Floating-Point</td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>0101</td>
<td>0010</td>
<td>vsetflp</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>Vector Z Raised to the Exponent Estimate Floating-Point</td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>0100</td>
<td>0010</td>
<td>vlogflp</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>Vector Log Base Z Estimate Floating-Point</td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>0000</td>
<td>0010</td>
<td>vrfinflp</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>Vector Round to Floating-Point Integral Nearest</td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>0001</td>
<td>0010</td>
<td>vrfzflp</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>Vector Round to Floating-Point Integral toward Zero</td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>0011</td>
<td>0010</td>
<td>vrfpflp</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>Vector Round to Floating-Point Integral toward Infinity</td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>0010</td>
<td>0010</td>
<td>vsbox</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>Vector AES S-Box</td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>0000</td>
<td>0010</td>
<td>vncipher</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>Vector AES Cipher Last</td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>0001</td>
<td>0010</td>
<td>vncipher</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>Vector AES Inverse Cipher Last</td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>0110</td>
<td>0010</td>
<td>vdfux</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>Vector Convert with round to nearest Signed Word format to FP</td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>0111</td>
<td>0010</td>
<td>vddfxx</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>Vector Convert with round to zero Signed Word format to FP Saturate</td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>0110</td>
<td>0010</td>
<td>vddfux</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>Vector Convert with round to zero FP To Signed Word format Saturate</td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>0111</td>
<td>0010</td>
<td>vddfxs</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>Vector Convert with round to zero FP To Signed Word format Saturate</td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>0100</td>
<td>0010</td>
<td>vnmxdp</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>Vector Maximum Floating-Point</td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>0101</td>
<td>0010</td>
<td>vnmnlp</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>Vector Minimum Floating-Point</td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>0010</td>
<td>0110</td>
<td>vmrgsub</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>Vector Merge High Byte</td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>0011</td>
<td>0110</td>
<td>vmrgshh</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>Vector Merge High Halfword</td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>0010</td>
<td>0110</td>
<td>vmrgshw</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>Vector Merge High Word</td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>0011</td>
<td>0110</td>
<td>vmrgshb</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>Vector Merge Low Byte</td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>0010</td>
<td>0110</td>
<td>vmrgshl</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>Vector Merge Low Halfword</td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>0011</td>
<td>0110</td>
<td>vmrgshw</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>Vector Merge Low Word</td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>0100</td>
<td>0110</td>
<td>vspitb</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>Vector Splat Byte</td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>0101</td>
<td>0110</td>
<td>vspith</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>Vector Splat Halfword</td>
</tr>
</tbody>
</table>

Figure 88. Power ISA AS Instruction Set Sorted by Opcode (Sheet 4 of 18)
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>Book</th>
<th>Page</th>
<th>Mnemonic</th>
<th>Version</th>
<th>Privilege</th>
<th>Mode Op</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>000100</td>
<td>I</td>
<td>I</td>
<td>258</td>
<td>vspltw</td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td>I/I</td>
<td>I</td>
<td>259</td>
<td>vspltsib</td>
<td>v2.03</td>
<td>Vector</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td>I/I/I</td>
<td>I/I</td>
<td>259</td>
<td>vsplshb</td>
<td>v2.03</td>
<td>Vector</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td>I/I/I/I</td>
<td>I/I/I</td>
<td>259</td>
<td>vsplshb</td>
<td>v2.03</td>
<td>Vector</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td>I</td>
<td>I</td>
<td>264</td>
<td>vsio</td>
<td>v2.03</td>
<td>Vector</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td>I</td>
<td>I</td>
<td>264</td>
<td>vars</td>
<td>v2.03</td>
<td>Vector</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td>I</td>
<td>I</td>
<td>339</td>
<td>vgblck</td>
<td>v2.07</td>
<td>Vector</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td>I</td>
<td>I</td>
<td>346</td>
<td>vpermq</td>
<td>v2.07</td>
<td>Vector</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td>I</td>
<td>I</td>
<td>346</td>
<td>vpermq</td>
<td>v2.07</td>
<td>Vector</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td>I</td>
<td>I</td>
<td>346</td>
<td>vpermq</td>
<td>v2.07</td>
<td>Vector</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td>I</td>
<td>I</td>
<td>346</td>
<td>vpermq</td>
<td>v2.07</td>
<td>Vector</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td>I</td>
<td>I</td>
<td>346</td>
<td>vpermq</td>
<td>v2.07</td>
<td>Vector</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td>I</td>
<td>I</td>
<td>346</td>
<td>vpermq</td>
<td>v2.07</td>
<td>Vector</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td>I</td>
<td>I</td>
<td>346</td>
<td>vpermq</td>
<td>v2.07</td>
<td>Vector</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td>I</td>
<td>I</td>
<td>346</td>
<td>vpermq</td>
<td>v2.07</td>
<td>Vector</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td>I</td>
<td>I</td>
<td>346</td>
<td>vpermq</td>
<td>v2.07</td>
<td>Vector</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td>I</td>
<td>I</td>
<td>346</td>
<td>vpermq</td>
<td>v2.07</td>
<td>Vector</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td>I</td>
<td>I</td>
<td>346</td>
<td>vpermq</td>
<td>v2.07</td>
<td>Vector</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td>I</td>
<td>I</td>
<td>346</td>
<td>vpermq</td>
<td>v2.07</td>
<td>Vector</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td>I</td>
<td>I</td>
<td>346</td>
<td>vpermq</td>
<td>v2.07</td>
<td>Vector</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td>I</td>
<td>I</td>
<td>346</td>
<td>vpermq</td>
<td>v2.07</td>
<td>Vector</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td>I</td>
<td>I</td>
<td>346</td>
<td>vpermq</td>
<td>v2.07</td>
<td>Vector</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td>I</td>
<td>I</td>
<td>346</td>
<td>vpermq</td>
<td>v2.07</td>
<td>Vector</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td>I</td>
<td>I</td>
<td>346</td>
<td>vpermq</td>
<td>v2.07</td>
<td>Vector</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td>I</td>
<td>I</td>
<td>346</td>
<td>vpermq</td>
<td>v2.07</td>
<td>Vector</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td>I</td>
<td>I</td>
<td>346</td>
<td>vpermq</td>
<td>v2.07</td>
<td>Vector</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td>I</td>
<td>I</td>
<td>346</td>
<td>vpermq</td>
<td>v2.07</td>
<td>Vector</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td>I</td>
<td>I</td>
<td>346</td>
<td>vpermq</td>
<td>v2.07</td>
<td>Vector</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td>I</td>
<td>I</td>
<td>346</td>
<td>vpermq</td>
<td>v2.07</td>
<td>Vector</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td>I</td>
<td>I</td>
<td>346</td>
<td>vpermq</td>
<td>v2.07</td>
<td>Vector</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 88. Power ISA AS Instruction Set Sorted by Opcode (Sheet 5 of 18)
<table>
<thead>
<tr>
<th>Instruction1</th>
<th>Format</th>
<th>Book</th>
<th>Page</th>
<th>Mnemonic</th>
<th>Version2</th>
<th>Privilege2</th>
<th>Mode Dep4</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>000100</td>
<td>VA</td>
<td>261</td>
<td>vsel</td>
<td>v2.03</td>
<td>Vector Select</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td>VA</td>
<td>260</td>
<td>vperm</td>
<td>v2.03</td>
<td>Vector Permute</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td>VA</td>
<td>263</td>
<td>voldo</td>
<td>v2.03</td>
<td>Vector Shift Left Double by Octet Immediate</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td>VA</td>
<td>338</td>
<td>vpermxor</td>
<td>v2.07</td>
<td>Vector Permute &amp; Exclusive-OR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011100</td>
<td>VA</td>
<td>322</td>
<td>vmaddfp</td>
<td>v2.03</td>
<td>Vector Multiply-Add Floating-Point</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010100</td>
<td>VA</td>
<td>322</td>
<td>vnmuldp</td>
<td>v2.03</td>
<td>Vector Negative Multiply-Subtract Floating-Point</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010100</td>
<td>VA</td>
<td>180</td>
<td>maddhdu</td>
<td>v3.0</td>
<td>Multiply-Add High Doubleword</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010100</td>
<td>VA</td>
<td>180</td>
<td>maddhdu</td>
<td>v3.0</td>
<td>Multiply-Add High Doubleword Unsigned</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010100</td>
<td>VA</td>
<td>180</td>
<td>maddhdu</td>
<td>v3.0</td>
<td>Multiply-Add High Doubleword Signed</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010100</td>
<td>VA</td>
<td>180</td>
<td>maddhdu</td>
<td>v3.0</td>
<td>Multiply-Add Low Doubleword</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010100</td>
<td>VA</td>
<td>180</td>
<td>maddhdu</td>
<td>v3.0</td>
<td>Multiply-Add Low Doubleword Unsigned</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010100</td>
<td>VA</td>
<td>180</td>
<td>maddhdu</td>
<td>v3.0</td>
<td>Multiply-Add Low Doubleword Signed</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010100</td>
<td>VA</td>
<td>180</td>
<td>maddhdu</td>
<td>v3.0</td>
<td>Multiply-Add Low Doubleword</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010100</td>
<td>VA</td>
<td>180</td>
<td>maddhdu</td>
<td>v3.0</td>
<td>Multiply-Add Low Doubleword Unsigned</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010100</td>
<td>VA</td>
<td>180</td>
<td>maddhdu</td>
<td>v3.0</td>
<td>Multiply-Add Low Doubleword Signed</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010100</td>
<td>VA</td>
<td>180</td>
<td>maddhdu</td>
<td>v3.0</td>
<td>Multiply-Add Low Doubleword</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010100</td>
<td>VA</td>
<td>180</td>
<td>maddhdu</td>
<td>v3.0</td>
<td>Multiply-Add Low Doubleword Unsigned</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010100</td>
<td>VA</td>
<td>180</td>
<td>maddhdu</td>
<td>v3.0</td>
<td>Multiply-Add Low Doubleword Signed</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010100</td>
<td>VA</td>
<td>180</td>
<td>maddhdu</td>
<td>v3.0</td>
<td>Multiply-Add Low Doubleword</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010100</td>
<td>VA</td>
<td>180</td>
<td>maddhdu</td>
<td>v3.0</td>
<td>Multiply-Add Low Doubleword Unsigned</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010100</td>
<td>VA</td>
<td>180</td>
<td>maddhdu</td>
<td>v3.0</td>
<td>Multiply-Add Low Doubleword Signed</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010100</td>
<td>VA</td>
<td>180</td>
<td>maddhdu</td>
<td>v3.0</td>
<td>Multiply-Add Low Doubleword</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 88. Power ISA AS Instruction Set Sorted by Opcode (Sheet 6 of 18)
<table>
<thead>
<tr>
<th>Instruction1</th>
<th>Format</th>
<th>Book</th>
<th>Page</th>
<th>Mnemonic</th>
<th>Version</th>
<th>Privilege3</th>
<th>Mode Dep4</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MD</td>
<td>106</td>
<td>.</td>
<td>hvic[]</td>
<td>PPC</td>
<td>SR</td>
<td></td>
<td>Rotate Left Doubleword Immediate then Clear Right</td>
</tr>
<tr>
<td></td>
<td>MD</td>
<td>105</td>
<td>.</td>
<td>hild[]</td>
<td>PPC</td>
<td>SR</td>
<td></td>
<td>Rotate Left Doubleword Immediate then Clear</td>
</tr>
<tr>
<td></td>
<td>MD</td>
<td>106</td>
<td>.</td>
<td>hidd[]</td>
<td>PPC</td>
<td>SR</td>
<td></td>
<td>Rotate Left Doubleword Immediate then Mask Insert</td>
</tr>
<tr>
<td></td>
<td>MDS</td>
<td>104</td>
<td>.</td>
<td>hidd[]</td>
<td>PPC</td>
<td>SR</td>
<td></td>
<td>Rotate Left Doubleword then Clear Left</td>
</tr>
<tr>
<td></td>
<td>MDS</td>
<td>104</td>
<td>.</td>
<td>hidd[]</td>
<td>PPC</td>
<td>SR</td>
<td></td>
<td>Rotate Left Doubleword then Clear</td>
</tr>
<tr>
<td></td>
<td>MD</td>
<td>85</td>
<td>.</td>
<td>cmp[]</td>
<td>P1</td>
<td></td>
<td></td>
<td>Compare</td>
</tr>
<tr>
<td></td>
<td>MD</td>
<td>85</td>
<td>.</td>
<td>cmpl[]</td>
<td>P1</td>
<td></td>
<td></td>
<td>Compare Logical</td>
</tr>
<tr>
<td></td>
<td>VX</td>
<td>122</td>
<td>setb[]</td>
<td>Set Boolean                                   v3.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MD</td>
<td>87</td>
<td>.</td>
<td>cmprb[]</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>Compare Ranged Byte</td>
</tr>
<tr>
<td></td>
<td>MD</td>
<td>88</td>
<td>.</td>
<td>cmpeqb[]</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>Compare Equal Byte</td>
</tr>
<tr>
<td></td>
<td>MD</td>
<td>120</td>
<td>.</td>
<td>mcovx[]</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>Move XER to CR Extended</td>
</tr>
<tr>
<td></td>
<td>MD</td>
<td>90</td>
<td>.</td>
<td>tw[]</td>
<td>P1</td>
<td></td>
<td></td>
<td>Trap Word</td>
</tr>
<tr>
<td></td>
<td>MD</td>
<td>91</td>
<td>.</td>
<td>td[]</td>
<td>PPC</td>
<td></td>
<td></td>
<td>Trap Doubleword</td>
</tr>
<tr>
<td></td>
<td>MD</td>
<td>247</td>
<td>.</td>
<td>lvxl[]</td>
<td>v2.0</td>
<td></td>
<td></td>
<td>Load Vector for Shift Left</td>
</tr>
<tr>
<td></td>
<td>MD</td>
<td>247</td>
<td>.</td>
<td>lvxm[]</td>
<td>v2.0</td>
<td></td>
<td></td>
<td>Load Vector for Shift Right</td>
</tr>
<tr>
<td></td>
<td>MD</td>
<td>860</td>
<td>.</td>
<td>lwat[]</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>Load Word Atomic</td>
</tr>
<tr>
<td></td>
<td>MD</td>
<td>860</td>
<td>.</td>
<td>ldat[]</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>Load Doubleword Atomic</td>
</tr>
<tr>
<td></td>
<td>MD</td>
<td>862</td>
<td>.</td>
<td>stdat[]</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>Store Word Atomic</td>
</tr>
<tr>
<td></td>
<td>MD</td>
<td>862</td>
<td>.</td>
<td>stdat[]</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>Store Doubleword Atomic</td>
</tr>
<tr>
<td></td>
<td>MD</td>
<td>855</td>
<td>.</td>
<td>copy[]</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>Copy</td>
</tr>
<tr>
<td></td>
<td>MD</td>
<td>856</td>
<td>.</td>
<td>cp_abort[]</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>CP_Abort</td>
</tr>
<tr>
<td></td>
<td>MD</td>
<td>855</td>
<td>.</td>
<td>paste[]</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>Paste</td>
</tr>
<tr>
<td></td>
<td>MD</td>
<td>242</td>
<td>.</td>
<td>tebx[]</td>
<td>v2.0</td>
<td></td>
<td></td>
<td>Load Vector Element Byte Indexed</td>
</tr>
<tr>
<td></td>
<td>MD</td>
<td>242</td>
<td>.</td>
<td>tvebx[]</td>
<td>v2.0</td>
<td></td>
<td></td>
<td>Load Vector Element Halfword Indexed</td>
</tr>
<tr>
<td></td>
<td>MD</td>
<td>243</td>
<td>.</td>
<td>tvewx[]</td>
<td>v2.0</td>
<td></td>
<td></td>
<td>Load Vector Element Word Indexed</td>
</tr>
<tr>
<td></td>
<td>MD</td>
<td>243</td>
<td>.</td>
<td>tix[]</td>
<td>v2.0</td>
<td></td>
<td></td>
<td>Load Vector Indexed</td>
</tr>
<tr>
<td></td>
<td>MD</td>
<td>245</td>
<td>.</td>
<td>stvebx[]</td>
<td>v2.0</td>
<td></td>
<td></td>
<td>Store Vector Element Byte Indexed</td>
</tr>
<tr>
<td></td>
<td>MD</td>
<td>245</td>
<td>.</td>
<td>stvehx[]</td>
<td>v2.0</td>
<td></td>
<td></td>
<td>Store Vector Element Halfword Indexed</td>
</tr>
<tr>
<td></td>
<td>MD</td>
<td>246</td>
<td>.</td>
<td>stvwx[]</td>
<td>v2.0</td>
<td></td>
<td></td>
<td>Store Vector Element Word Indexed</td>
</tr>
<tr>
<td></td>
<td>MD</td>
<td>246</td>
<td>.</td>
<td>stvx[]</td>
<td>v2.0</td>
<td></td>
<td></td>
<td>Store Vector Indexed</td>
</tr>
<tr>
<td></td>
<td>MD</td>
<td>246</td>
<td>.</td>
<td>tx[]</td>
<td>v2.0</td>
<td></td>
<td></td>
<td>Load Vector Indexed Last</td>
</tr>
<tr>
<td></td>
<td>MD</td>
<td>246</td>
<td>.</td>
<td>stvl[]</td>
<td>v2.0</td>
<td></td>
<td></td>
<td>Store Vector Indexed Last</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>70</td>
<td>.</td>
<td>subf[c][]</td>
<td>P1</td>
<td>SR</td>
<td></td>
<td>Subtract From Carrying</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>69</td>
<td>.</td>
<td>subf[]</td>
<td>PPC</td>
<td>SR</td>
<td></td>
<td>Subtract From</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>72</td>
<td>.</td>
<td>negf[]</td>
<td>P1</td>
<td>SR</td>
<td></td>
<td>Negate</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>71</td>
<td>.</td>
<td>subf[]</td>
<td>P1</td>
<td>SR</td>
<td></td>
<td>Subtract From Extended</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>72</td>
<td>.</td>
<td>subfze[]</td>
<td>P1</td>
<td>SR</td>
<td></td>
<td>Subtract From Zero Extended</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>71</td>
<td>.</td>
<td>subfme[]</td>
<td>P1</td>
<td>SR</td>
<td></td>
<td>Subtract From Minus One Extended</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>73</td>
<td>.</td>
<td>multd[]</td>
<td>PPC</td>
<td>SR</td>
<td></td>
<td>Multiply High Doubleword Unsigned</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>79</td>
<td>.</td>
<td>multd[]</td>
<td>PPC</td>
<td>SR</td>
<td></td>
<td>Multiply High Doubleword</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>79</td>
<td>.</td>
<td>multd[]</td>
<td>PPC</td>
<td>SR</td>
<td></td>
<td>Multiply High Doubleword Unsigned</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>81</td>
<td>.</td>
<td>divid[]</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>Modulo Signed Doubleword</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>82</td>
<td>.</td>
<td>divid[]</td>
<td>v2.0</td>
<td></td>
<td></td>
<td>Divide Doubleword Extended Unsighed</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>82</td>
<td>.</td>
<td>divid[]</td>
<td>v2.0</td>
<td></td>
<td></td>
<td>Divide Doubleword Extended</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>81</td>
<td>.</td>
<td>divid[]</td>
<td>PPC</td>
<td>SR</td>
<td></td>
<td>Divide Doubleword Unsighed</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>81</td>
<td>.</td>
<td>divid[]</td>
<td>PPC</td>
<td>SR</td>
<td></td>
<td>Divide Doubleword</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>83</td>
<td>.</td>
<td>modsd[]</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>Modulo Signed Doubleword</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>70</td>
<td>.</td>
<td>addc[]</td>
<td>P1</td>
<td>SR</td>
<td></td>
<td>Add Carrying</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>111</td>
<td>.</td>
<td>addc[]</td>
<td>v2.0</td>
<td></td>
<td></td>
<td>Add &amp; Generate Slices</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>71</td>
<td>.</td>
<td>addc[]</td>
<td>P1</td>
<td>SR</td>
<td></td>
<td>Add Extended</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>72</td>
<td>.</td>
<td>addc[]</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>Add Extended using alternate carry</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>72</td>
<td>.</td>
<td>addc[]</td>
<td>P1</td>
<td>SR</td>
<td></td>
<td>Add to Zero Extended</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>73</td>
<td>.</td>
<td>addc[]</td>
<td>P1</td>
<td>SR</td>
<td></td>
<td>Add to Minus One Extended</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>73</td>
<td>.</td>
<td>mulhw[]</td>
<td>PPC</td>
<td>SR</td>
<td></td>
<td>Multiply High Word Unsighed</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>73</td>
<td>.</td>
<td>mulhw[]</td>
<td>PPC</td>
<td>SR</td>
<td></td>
<td>Multiply High Word</td>
</tr>
</tbody>
</table>

Figure 88. Power ISA AS Instruction Set Sorted by Opcode (Sheet 7 of 18)
<table>
<thead>
<tr>
<th>Instruction(^1)</th>
<th>Format</th>
<th>Book</th>
<th>Page</th>
<th>Mnemonic</th>
<th>Version(^2)</th>
<th>Privilege(^3)</th>
<th>Mode Dep(^4)</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>011111</td>
<td>011111</td>
<td>XO</td>
<td>73</td>
<td>mulw[0]</td>
<td>P1</td>
<td>SR</td>
<td></td>
<td>Multiply Low Word</td>
</tr>
<tr>
<td>X111111</td>
<td>010000</td>
<td>X</td>
<td>77</td>
<td>moduw</td>
<td>v3.0</td>
<td></td>
<td>Modulo Signed Word</td>
<td></td>
</tr>
<tr>
<td>X111111</td>
<td>010000</td>
<td>X</td>
<td>75</td>
<td>divwu[0]</td>
<td>v2.06</td>
<td>SR</td>
<td>Divide Word Extended Unsigned</td>
<td></td>
</tr>
<tr>
<td>X111111</td>
<td>010000</td>
<td>X</td>
<td>75</td>
<td>divw[0]</td>
<td>v2.06</td>
<td>SR</td>
<td>Divide Word Extended</td>
<td></td>
</tr>
<tr>
<td>X111111</td>
<td>010000</td>
<td>X</td>
<td>74</td>
<td>divw[0]</td>
<td>PPC</td>
<td>SR</td>
<td>Divide Word Unsigned</td>
<td></td>
</tr>
<tr>
<td>X111111</td>
<td>010000</td>
<td>X</td>
<td>74</td>
<td>divw[0]</td>
<td>PPC</td>
<td>SR</td>
<td>Divide Word</td>
<td></td>
</tr>
<tr>
<td>X111111</td>
<td>010000</td>
<td>X</td>
<td>77</td>
<td>modsw</td>
<td>v3.0</td>
<td></td>
<td>Modulo Signed Word</td>
<td></td>
</tr>
<tr>
<td>X111111</td>
<td>010000</td>
<td>X</td>
<td>484</td>
<td>liwswx</td>
<td>v2.07</td>
<td>Load VSX Scalar as Integer Word &amp; Zero Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X111111</td>
<td>010000</td>
<td>X</td>
<td>483</td>
<td>liwswx</td>
<td>v2.07</td>
<td>Load VSX Scalar as Integer Word Algebraic Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X111111</td>
<td>010000</td>
<td>X</td>
<td>500</td>
<td>stxswx</td>
<td>v2.07</td>
<td>Store VSX Scalar as Integer Word Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X111111</td>
<td>010000</td>
<td>X</td>
<td>492</td>
<td>lvv</td>
<td>v3.0</td>
<td>Load VSX Vector Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X111111</td>
<td>010000</td>
<td>X</td>
<td>494</td>
<td>lvwdx</td>
<td>v2.06</td>
<td>Load VSX Vector Doubleword &amp; Splat Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X111111</td>
<td>010000</td>
<td>X</td>
<td>497</td>
<td>lvxswx</td>
<td>v3.0</td>
<td>Load VSX Vector Word &amp; Splat Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X111111</td>
<td>010000</td>
<td>X</td>
<td>510</td>
<td>stxv</td>
<td>v3.0</td>
<td>Store VSX Vector Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X111111</td>
<td>010000</td>
<td>X</td>
<td>485</td>
<td>liwspx</td>
<td>v2.07</td>
<td>Load VSX Scalar Single-Precision Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X111111</td>
<td>010000</td>
<td>X</td>
<td>488</td>
<td>liwdx</td>
<td>v2.06</td>
<td>Load VSX Scalar Doubleword Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X111111</td>
<td>010000</td>
<td>X</td>
<td>502</td>
<td>stxspx</td>
<td>v2.07</td>
<td>Load VSX Scalar Single-Precision Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X111111</td>
<td>010000</td>
<td>X</td>
<td>498</td>
<td>stxspdx</td>
<td>v2.06</td>
<td>Store VSX Scalar Doubleword Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X111111</td>
<td>010000</td>
<td>X</td>
<td>496</td>
<td>lvxw4x</td>
<td>v2.06</td>
<td>Load VSX Vector Word*4 Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X111111</td>
<td>010000</td>
<td>X</td>
<td>495</td>
<td>lvxhix</td>
<td>v3.0</td>
<td>Load VSX Vector Halfword*8 Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X111111</td>
<td>010000</td>
<td>X</td>
<td>497</td>
<td>lvx8x</td>
<td>v3.0</td>
<td>Load VSX Vector Doubleword*2 Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X111111</td>
<td>010000</td>
<td>X</td>
<td>487</td>
<td>lvx16x</td>
<td>v3.0</td>
<td>Load VSX Vector Byte*16 Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X111111</td>
<td>010000</td>
<td>X</td>
<td>501</td>
<td>stxv4x</td>
<td>v2.06</td>
<td>Store VSX Vector Word*4 Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X111111</td>
<td>010000</td>
<td>X</td>
<td>501</td>
<td>stxv4x</td>
<td>v2.06</td>
<td>Store VSX Vector Doubleword*2 Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X111111</td>
<td>010000</td>
<td>X</td>
<td>501</td>
<td>stxv4x</td>
<td>v2.06</td>
<td>Store VSX Vector Doubleword*2 Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X111111</td>
<td>010000</td>
<td>X</td>
<td>498</td>
<td>lvw</td>
<td>v3.0</td>
<td>Load VSX Vector with Length</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X111111</td>
<td>010000</td>
<td>X</td>
<td>491</td>
<td>lvx</td>
<td>v3.0</td>
<td>Load VSX Vector Left-justified with Length</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X111111</td>
<td>010000</td>
<td>X</td>
<td>507</td>
<td>stxv</td>
<td>v3.0</td>
<td>Store VSX Vector with Length</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X111111</td>
<td>010000</td>
<td>X</td>
<td>509</td>
<td>stxv</td>
<td>v3.0</td>
<td>Store VSX Vector Left-justified with Length</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X111111</td>
<td>010000</td>
<td>X</td>
<td>482</td>
<td>liwibx</td>
<td>v3.0</td>
<td>Load VSX Scalar as Integer Byte &amp; Zero Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X111111</td>
<td>010000</td>
<td>X</td>
<td>482</td>
<td>liwibx</td>
<td>v3.0</td>
<td>Load VSX Scalar as Integer Halfword &amp; Zero Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X111111</td>
<td>010000</td>
<td>X</td>
<td>499</td>
<td>stxibx</td>
<td>v3.0</td>
<td>Store VSX Scalar as Integer Byte Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X111111</td>
<td>010000</td>
<td>X</td>
<td>499</td>
<td>stxibx</td>
<td>v3.0</td>
<td>Store VSX Scalar as Integer Halfword Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X111111</td>
<td>010000</td>
<td>X</td>
<td>1131</td>
<td>msgsndp</td>
<td>v2.07</td>
<td>Message Send Privileged</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X111111</td>
<td>010000</td>
<td>X</td>
<td>1132</td>
<td>msgndip</td>
<td>v2.07</td>
<td>Message Clear Privileged</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X111111</td>
<td>010000</td>
<td>X</td>
<td>1129</td>
<td>msgnd</td>
<td>v2.07</td>
<td>Message Send</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X111111</td>
<td>010000</td>
<td>X</td>
<td>1130</td>
<td>msgdor</td>
<td>v2.07</td>
<td>Message Clear</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X111111</td>
<td>010000</td>
<td>X</td>
<td>908</td>
<td>mthrbr</td>
<td>v2.07</td>
<td>Move From BHRB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X111111</td>
<td>010000</td>
<td>X</td>
<td>908</td>
<td>mthrbr</td>
<td>v2.07</td>
<td>Move From BHRB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X111111</td>
<td>010000</td>
<td>X</td>
<td>908</td>
<td>mthrbr</td>
<td>v2.07</td>
<td>Clear BHRB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X111111</td>
<td>010000</td>
<td>X</td>
<td>891</td>
<td>tend</td>
<td>v2.07</td>
<td>Transaction End &amp; record</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X111111</td>
<td>010000</td>
<td>X</td>
<td>895</td>
<td>tcheck</td>
<td>v2.07</td>
<td>Transaction Check &amp; record</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X111111</td>
<td>010000</td>
<td>X</td>
<td>895</td>
<td>tsw</td>
<td>v2.07</td>
<td>Transaction Suspend or Resume &amp; record</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X111111</td>
<td>010000</td>
<td>X</td>
<td>890</td>
<td>tbegin</td>
<td>v2.07</td>
<td>Transaction Begin &amp; record</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X111111</td>
<td>010000</td>
<td>X</td>
<td>893</td>
<td>tabortwc</td>
<td>v2.07</td>
<td>Transaction Abort Word Conditional &amp; record</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X111111</td>
<td>010000</td>
<td>X</td>
<td>894</td>
<td>tabortdc</td>
<td>v2.07</td>
<td>Transaction Abort Doubleword Conditional &amp; record</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X111111</td>
<td>010000</td>
<td>X</td>
<td>893</td>
<td>tabortwcl</td>
<td>v2.07</td>
<td>Transaction Abort Word Conditional Immediate &amp; record</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X111111</td>
<td>010000</td>
<td>X</td>
<td>894</td>
<td>tabortwc</td>
<td>v2.07</td>
<td>Transaction Abort Doubleword Conditional Immediate &amp; record</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X111111</td>
<td>010000</td>
<td>X</td>
<td>892</td>
<td>tabort</td>
<td>v2.07</td>
<td>Transaction Abort &amp; record</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X111111</td>
<td>010000</td>
<td>X</td>
<td>969</td>
<td>treclaim</td>
<td>v2.07</td>
<td>Transaction Reclaim &amp; record</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X111111</td>
<td>010000</td>
<td>X</td>
<td>970</td>
<td>trechkpt</td>
<td>v2.07</td>
<td>Transaction Recheckpoint &amp; record</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X111111</td>
<td>010000</td>
<td>X</td>
<td>91</td>
<td>tsel</td>
<td>v2.03</td>
<td>Integer Select</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X111111</td>
<td>010000</td>
<td>X</td>
<td>121</td>
<td>mtrf</td>
<td>P1</td>
<td>Move To CR Fields</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X111111</td>
<td>010000</td>
<td>X</td>
<td>121</td>
<td>mtrcfrf</td>
<td>v2.01</td>
<td>Move To One CR Field</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X111111</td>
<td>010000</td>
<td>X</td>
<td>977</td>
<td>mtrmr</td>
<td>P1</td>
<td>Move To MSR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X111111</td>
<td>010000</td>
<td>X</td>
<td>978</td>
<td>mtrmsrd</td>
<td>PPC</td>
<td>Move To MSR Doubleword</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 88. Power ISA AS Instruction Set Sorted by Opcode (Sheet 8 of 18)
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>Book</th>
<th>Page</th>
<th>Mnemonic</th>
<th>Version</th>
<th>Privilege</th>
<th>Mode</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:5</td>
<td>6:10</td>
<td>11:15</td>
<td>16:20</td>
<td>21:25</td>
<td>26:31</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1038</td>
<td>tbiel</td>
<td>v2.03</td>
<td>P</td>
<td>64</td>
<td>TLB Invalidate Entry Local</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1034</td>
<td>tbie</td>
<td>P1</td>
<td>HV</td>
<td>64</td>
<td>TLB Invalidate Entry</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1032</td>
<td>sbis</td>
<td>v3.0</td>
<td>P</td>
<td>SLB Synchronize</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1029</td>
<td>sbmte</td>
<td>v2.00</td>
<td>P</td>
<td>SLB Move To Entry</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1024</td>
<td>sbie</td>
<td>PPC</td>
<td>P</td>
<td>SLB Invalidate Entry</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1025</td>
<td>sbieg</td>
<td>v3.0</td>
<td>P</td>
<td>SLB Invalidate Entry Global</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1026</td>
<td>sbiea</td>
<td>PPC</td>
<td>P</td>
<td>SLB Invalidate All</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1028</td>
<td>sbieg</td>
<td>v3.0</td>
<td>P</td>
<td>SLB Invalidate All Global</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>122</td>
<td>mfsr</td>
<td>P1</td>
<td>Move From CR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>113</td>
<td>mfvarw</td>
<td>v2.07</td>
<td>P</td>
<td>Move From VSR Doubleword</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>114</td>
<td>mfvarw</td>
<td>v2.07</td>
<td>P</td>
<td>Move To VSR Doubleword</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>115</td>
<td>mfvarw</td>
<td>v2.07</td>
<td>P</td>
<td>Move To VSR Word &amp; Zero</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>115</td>
<td>mtvvarw</td>
<td>v2.07</td>
<td>P</td>
<td>Move To VSR Word Algebraic</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>117</td>
<td>mfvarw</td>
<td>v2.07</td>
<td>P</td>
<td>Move To VSR Word &amp; Zero</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>975</td>
<td>mfsr</td>
<td>P1</td>
<td>O</td>
<td>Move From SPR</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>116</td>
<td>mtvvarw</td>
<td>v3.0</td>
<td>P</td>
<td>Move To VSR Word &amp; Splat</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>115</td>
<td>mtvvarw</td>
<td>v3.0</td>
<td>P</td>
<td>Move To VSR Doubleword</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>117</td>
<td>mfsr</td>
<td>P1</td>
<td>O</td>
<td>Move To SPR</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1300</td>
<td>sbmte</td>
<td>v2.00</td>
<td>P</td>
<td>SLB Move From Entry VSID</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1031</td>
<td>sbmfe</td>
<td>v2.00</td>
<td>P</td>
<td>SLB Move From Entry ESIID</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1031</td>
<td>sbf</td>
<td>v2.05</td>
<td>P</td>
<td>SR</td>
<td>SLB Find Entry ESIID &amp; record</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>865</td>
<td>lwax</td>
<td>PPC</td>
<td>P</td>
<td>Load Word &amp; Reserve Indexed</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>864</td>
<td>lbaru</td>
<td>v2.06</td>
<td>P</td>
<td>Load Byte &amp; Reserve Indexed</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>869</td>
<td>ldaa</td>
<td>PPC</td>
<td>Load Doubleword And Reserve indexed</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>865</td>
<td>lbaru</td>
<td>v2.06</td>
<td>P</td>
<td>Load Halfword And Reserve Indexed XForm</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>871</td>
<td>lbaru</td>
<td>v2.07</td>
<td>P</td>
<td>Load Quadword And Reserve Indexed</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>61</td>
<td>ltcbrx</td>
<td>v2.06</td>
<td>P</td>
<td>Load Doubleword Byte-Reverse Indexed</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>61</td>
<td>stdbrx</td>
<td>v2.06</td>
<td>P</td>
<td>Store Doubleword Byte-Reverse Indexed</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>53</td>
<td>ldx</td>
<td>PPC</td>
<td>Load Doubleword Indexed</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>53</td>
<td>ldux</td>
<td>PPC</td>
<td>Load Doubleword with Update Indexed</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>57</td>
<td>stdx</td>
<td>PPC</td>
<td>Store Doubleword Indexed</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>57</td>
<td>stdux</td>
<td>PPC</td>
<td>Store Doubleword with Update Indexed</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>52</td>
<td>lwax</td>
<td>PPC</td>
<td>P</td>
<td>Load Word Algebraic Indexed</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>52</td>
<td>lwaux</td>
<td>PPC</td>
<td>P</td>
<td>Load Word Algebraic with Update Indexed</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>lwax</td>
<td>P1</td>
<td>Load String Word Indexed</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>lswi</td>
<td>P1</td>
<td>Load String Word Immediate</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>65</td>
<td>lswax</td>
<td>P1</td>
<td>Store String Word Indexed</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>65</td>
<td>lswax</td>
<td>P1</td>
<td>Store String Word Indexed</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>966</td>
<td>lwzx</td>
<td>v2.05</td>
<td>HV</td>
<td>Load Word &amp; Zero Caching Inhibited Indexed</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>966</td>
<td>lhzx</td>
<td>v2.05</td>
<td>HV</td>
<td>Load Halfword &amp; Zero Caching Inhibited Indexed</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>966</td>
<td>lhzx</td>
<td>v2.05</td>
<td>HV</td>
<td>Load Halfword &amp; Zero Caching Inhibited Indexed</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>966</td>
<td>lhzx</td>
<td>v2.05</td>
<td>HV</td>
<td>Load Halfword &amp; Zero Caching Inhibited Indexed</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>967</td>
<td>stzox</td>
<td>v2.05</td>
<td>HV</td>
<td>Store Word Caching Inhibited Indexed</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>967</td>
<td>stzox</td>
<td>v2.05</td>
<td>HV</td>
<td>Store Halfword Caching Inhibited Indexed</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>967</td>
<td>stzox</td>
<td>v2.05</td>
<td>HV</td>
<td>Store Word Caching Inhibited Indexed</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>967</td>
<td>stzox</td>
<td>v2.05</td>
<td>HV</td>
<td>Store Halfword Caching Inhibited Indexed</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>967</td>
<td>stzox</td>
<td>v2.05</td>
<td>HV</td>
<td>Store Word Caching Inhibited Indexed</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>967</td>
<td>stzox</td>
<td>v2.05</td>
<td>HV</td>
<td>Store Halfword Caching Inhibited Indexed</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>840</td>
<td>icbt</td>
<td>v2.07</td>
<td>P</td>
<td>Instruction Cache Block Touch</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>851</td>
<td>dcbst</td>
<td>PPC</td>
<td>Data Cache Block Store</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>852</td>
<td>dcbf</td>
<td>PPC</td>
<td>Data Cache Block Flush</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>850</td>
<td>dcbt</td>
<td>PPC</td>
<td>Data Cache Block Touch for Store</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 88. Power ISA AS Instruction Set Sorted by Opcode (Sheet 9 of 18)
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>Book</th>
<th>Page</th>
<th>Mnemonic</th>
<th>Version2</th>
<th>Privilege3</th>
<th>Mode Dep4</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0:5</td>
<td>6:10</td>
<td>13:15</td>
<td>16:23</td>
<td>21:25</td>
<td>26:31</td>
<td></td>
<td></td>
</tr>
<tr>
<td>849</td>
<td>X</td>
<td>I</td>
<td>I</td>
<td>dcbt</td>
<td>PPC</td>
<td>P1</td>
<td></td>
<td>Data Cache Block Touch</td>
</tr>
<tr>
<td>60</td>
<td>X</td>
<td>I</td>
<td>I</td>
<td>lwbrx</td>
<td>PPC</td>
<td>P1</td>
<td></td>
<td>Load Word Byte-Reverse Indexed</td>
</tr>
<tr>
<td>1042</td>
<td>X</td>
<td>I</td>
<td>I</td>
<td>tbsync</td>
<td>PPC</td>
<td>HV/P</td>
<td></td>
<td>TLB Synchronize</td>
</tr>
<tr>
<td>873</td>
<td>X</td>
<td>I</td>
<td>I</td>
<td>sync</td>
<td>PPC</td>
<td>P1</td>
<td></td>
<td>Synchronize</td>
</tr>
<tr>
<td>80</td>
<td>X</td>
<td>I</td>
<td>I</td>
<td>stbrx</td>
<td>PPC</td>
<td>P1</td>
<td></td>
<td>Store Word Byte-Reverse Indexed</td>
</tr>
<tr>
<td>80</td>
<td>X</td>
<td>I</td>
<td>I</td>
<td>stbrx</td>
<td>PPC</td>
<td>P1</td>
<td></td>
<td>Load Halfword Byte-Reverse Indexed</td>
</tr>
<tr>
<td>875</td>
<td>X</td>
<td>I</td>
<td>I</td>
<td>elseo</td>
<td>PPC</td>
<td></td>
<td></td>
<td>Enforce In-order Execution of I/O</td>
</tr>
<tr>
<td>1153</td>
<td>X</td>
<td>I</td>
<td>I</td>
<td>msgsync</td>
<td>v3.0</td>
<td>HV</td>
<td></td>
<td>Message Synchronize</td>
</tr>
<tr>
<td>80</td>
<td>X</td>
<td>I</td>
<td>I</td>
<td>stbrx</td>
<td>PPC</td>
<td>P1</td>
<td></td>
<td>Store Halfword Byte-Reverse Indexed</td>
</tr>
<tr>
<td>840</td>
<td>X</td>
<td>I</td>
<td>I</td>
<td>icbi</td>
<td>PPC</td>
<td>P1</td>
<td></td>
<td>Instruction Cache Block Invalidate</td>
</tr>
<tr>
<td>867</td>
<td>X</td>
<td>I</td>
<td>I</td>
<td>sthcx.</td>
<td>PPC</td>
<td>P1</td>
<td></td>
<td>Store Halfword Conditional Indexed &amp; record</td>
</tr>
<tr>
<td>866</td>
<td>X</td>
<td>I</td>
<td>I</td>
<td>stbcx.</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>Store Byte Conditional Indexed &amp; record</td>
</tr>
<tr>
<td>869</td>
<td>X</td>
<td>I</td>
<td>I</td>
<td>stdcx.</td>
<td>PPC</td>
<td>P1</td>
<td></td>
<td>Store Doubleword Conditional Indexed &amp; record</td>
</tr>
<tr>
<td>868</td>
<td>X</td>
<td>I</td>
<td>I</td>
<td>stdcx.</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>Store Byte Conditional Indexed &amp; record</td>
</tr>
<tr>
<td>1109</td>
<td>X</td>
<td>I</td>
<td>I</td>
<td>stdcx.</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>Store Halfword Conditional Indexed &amp; record</td>
</tr>
<tr>
<td>867</td>
<td>X</td>
<td>I</td>
<td>I</td>
<td>sthcx.</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>Store Halfword Conditional Indexed &amp; record</td>
</tr>
<tr>
<td>0000</td>
<td>X</td>
<td>I</td>
<td>I</td>
<td>lwzx</td>
<td>P1</td>
<td></td>
<td></td>
<td>Load Word &amp; Zero Indexed</td>
</tr>
<tr>
<td>51</td>
<td>X</td>
<td>I</td>
<td>I</td>
<td>lwzx</td>
<td>P1</td>
<td></td>
<td></td>
<td>Load Word &amp; Zero with Update Indexed</td>
</tr>
<tr>
<td>48</td>
<td>X</td>
<td>I</td>
<td>I</td>
<td>tuxz</td>
<td>P1</td>
<td></td>
<td></td>
<td>Load Byte &amp; Zero Indexed</td>
</tr>
<tr>
<td>48</td>
<td>X</td>
<td>I</td>
<td>I</td>
<td>tuxz</td>
<td>P1</td>
<td></td>
<td></td>
<td>Load Byte &amp; Zero with Update Indexed</td>
</tr>
<tr>
<td>36</td>
<td>X</td>
<td>I</td>
<td>I</td>
<td>stwx</td>
<td>P1</td>
<td></td>
<td></td>
<td>Store Word Indexed</td>
</tr>
<tr>
<td>36</td>
<td>X</td>
<td>I</td>
<td>I</td>
<td>stwx</td>
<td>P1</td>
<td></td>
<td></td>
<td>Store Word with Update Indexed</td>
</tr>
<tr>
<td>54</td>
<td>X</td>
<td>I</td>
<td>I</td>
<td>stbx</td>
<td>P1</td>
<td></td>
<td></td>
<td>Store Byte Indexed</td>
</tr>
<tr>
<td>54</td>
<td>X</td>
<td>I</td>
<td>I</td>
<td>stbx</td>
<td>P1</td>
<td></td>
<td></td>
<td>Store Byte with Update Indexed</td>
</tr>
<tr>
<td>49</td>
<td>X</td>
<td>I</td>
<td>I</td>
<td>fzx</td>
<td>P1</td>
<td></td>
<td></td>
<td>Load Halfword &amp; Zero Indexed</td>
</tr>
<tr>
<td>49</td>
<td>X</td>
<td>I</td>
<td>I</td>
<td>fzx</td>
<td>P1</td>
<td></td>
<td></td>
<td>Load Halfword &amp; Zero with Update Indexed</td>
</tr>
<tr>
<td>50</td>
<td>X</td>
<td>I</td>
<td>I</td>
<td>fuxz</td>
<td>P1</td>
<td></td>
<td></td>
<td>Load Halfword Algebraic Indexed</td>
</tr>
<tr>
<td>50</td>
<td>X</td>
<td>I</td>
<td>I</td>
<td>fuxz</td>
<td>P1</td>
<td></td>
<td></td>
<td>Load Halfword Algebraic with Update Indexed</td>
</tr>
<tr>
<td>55</td>
<td>X</td>
<td>I</td>
<td>I</td>
<td>sthx</td>
<td>P1</td>
<td></td>
<td></td>
<td>Store Halfword Indexed</td>
</tr>
<tr>
<td>55</td>
<td>X</td>
<td>I</td>
<td>I</td>
<td>sthx</td>
<td>P1</td>
<td></td>
<td></td>
<td>Store Halfword with Update Indexed</td>
</tr>
<tr>
<td>1000</td>
<td>X</td>
<td>I</td>
<td>I</td>
<td>ifsxx</td>
<td>P1</td>
<td></td>
<td></td>
<td>Load Floating Single Indexed</td>
</tr>
<tr>
<td>141</td>
<td>X</td>
<td>I</td>
<td>I</td>
<td>ifsxx</td>
<td>P1</td>
<td></td>
<td></td>
<td>Load Floating Single with Update Indexed</td>
</tr>
<tr>
<td>142</td>
<td>X</td>
<td>I</td>
<td>I</td>
<td>ifsxx</td>
<td>P1</td>
<td></td>
<td></td>
<td>Load Floating Double Indexed</td>
</tr>
<tr>
<td>142</td>
<td>X</td>
<td>I</td>
<td>I</td>
<td>ifsxx</td>
<td>P1</td>
<td></td>
<td></td>
<td>Load Floating Double with Update Indexed</td>
</tr>
<tr>
<td>143</td>
<td>X</td>
<td>I</td>
<td>I</td>
<td>ftx</td>
<td>P1</td>
<td></td>
<td></td>
<td>Load Floating Single Indexed</td>
</tr>
<tr>
<td>143</td>
<td>X</td>
<td>I</td>
<td>I</td>
<td>ftx</td>
<td>P1</td>
<td></td>
<td></td>
<td>Load Floating Single with Update Indexed</td>
</tr>
<tr>
<td>144</td>
<td>X</td>
<td>I</td>
<td>I</td>
<td>ftx</td>
<td>P1</td>
<td></td>
<td></td>
<td>Load Floating Double Indexed</td>
</tr>
<tr>
<td>144</td>
<td>X</td>
<td>I</td>
<td>I</td>
<td>ftx</td>
<td>P1</td>
<td></td>
<td></td>
<td>Load Floating Double with Update Indexed</td>
</tr>
<tr>
<td>145</td>
<td>X</td>
<td>I</td>
<td>I</td>
<td>ftux</td>
<td>P1</td>
<td></td>
<td></td>
<td>Load Floating Double Indexed</td>
</tr>
<tr>
<td>145</td>
<td>X</td>
<td>I</td>
<td>I</td>
<td>ftux</td>
<td>P1</td>
<td></td>
<td></td>
<td>Load Floating Double with Update Indexed</td>
</tr>
<tr>
<td>146</td>
<td>X</td>
<td>I</td>
<td>I</td>
<td>ftux</td>
<td>P1</td>
<td></td>
<td></td>
<td>Load Floating Double Indexed</td>
</tr>
<tr>
<td>146</td>
<td>X</td>
<td>I</td>
<td>I</td>
<td>ftux</td>
<td>P1</td>
<td></td>
<td></td>
<td>Load Floating Double with Update Indexed</td>
</tr>
<tr>
<td>143</td>
<td>X</td>
<td>I</td>
<td>I</td>
<td>ftx</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>Load Floating Double Pair Indexed</td>
</tr>
<tr>
<td>143</td>
<td>X</td>
<td>I</td>
<td>I</td>
<td>ftx</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>Load Floating as Integer Word Algebraic Indexed</td>
</tr>
<tr>
<td>143</td>
<td>X</td>
<td>I</td>
<td>I</td>
<td>ftx</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>Load Floating as Integer Word &amp; Zero Indexed</td>
</tr>
<tr>
<td>149</td>
<td>X</td>
<td>I</td>
<td>I</td>
<td>sftdx</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>Store Floating Double Pair Indexed</td>
</tr>
<tr>
<td>147</td>
<td>X</td>
<td>I</td>
<td>I</td>
<td>sftdx</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>Store Floating as Integer Indexed</td>
</tr>
<tr>
<td>107</td>
<td>X</td>
<td>I</td>
<td>I</td>
<td>slw[ ]</td>
<td>P1</td>
<td>SR</td>
<td></td>
<td>Shift Left Word</td>
</tr>
<tr>
<td>107</td>
<td>X</td>
<td>I</td>
<td>I</td>
<td>srw[ ]</td>
<td>P1</td>
<td>SR</td>
<td></td>
<td>Shift Right Word</td>
</tr>
<tr>
<td>108</td>
<td>X</td>
<td>I</td>
<td>I</td>
<td>sraw[ ]</td>
<td>P1</td>
<td>SR</td>
<td></td>
<td>Shift Right Algebraic Word</td>
</tr>
<tr>
<td>108</td>
<td>X</td>
<td>I</td>
<td>I</td>
<td>sraw[ ]</td>
<td>P1</td>
<td>SR</td>
<td></td>
<td>Shift Right Algebraic Word Immediate</td>
</tr>
<tr>
<td>96</td>
<td>X</td>
<td>I</td>
<td>I</td>
<td>cnti[ ]</td>
<td>P1</td>
<td>SR</td>
<td></td>
<td>Count Leading Zeros Word</td>
</tr>
<tr>
<td>99</td>
<td>X</td>
<td>I</td>
<td>I</td>
<td>cnti[ ]</td>
<td>PPC</td>
<td>SR</td>
<td></td>
<td>Count Leading Zeros Doubleword</td>
</tr>
<tr>
<td>97</td>
<td>X</td>
<td>I</td>
<td>I</td>
<td>popcntb</td>
<td>v2.02</td>
<td></td>
<td></td>
<td>Population Count Byte</td>
</tr>
<tr>
<td>98</td>
<td>X</td>
<td>I</td>
<td>I</td>
<td>ptwy</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>Parity Word</td>
</tr>
<tr>
<td>98</td>
<td>X</td>
<td>I</td>
<td>I</td>
<td>ptwd</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>Parity Doubleword</td>
</tr>
<tr>
<td>111</td>
<td>X</td>
<td>I</td>
<td>I</td>
<td>cdlobd</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>Convert Decimals To Binary Coded Decimal</td>
</tr>
<tr>
<td>111</td>
<td>X</td>
<td>I</td>
<td>I</td>
<td>bcdlo</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>Convert Binary Coded Decimal To Decimals</td>
</tr>
<tr>
<td>97</td>
<td>X</td>
<td>I</td>
<td>I</td>
<td>popcntd</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>Population Count Words</td>
</tr>
<tr>
<td>99</td>
<td>X</td>
<td>I</td>
<td>I</td>
<td>popcntd</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>Population Count Doubleword</td>
</tr>
</tbody>
</table>

Figure 88. Power ISA AS Instruction Set Sorted by Opcode (Sheet 10 of 18)
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>Book</th>
<th>Page</th>
<th>Mnemonic</th>
<th>Version</th>
<th>Privilege</th>
<th>Mode</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>011111 1111111111010</td>
<td>X</td>
<td>96</td>
<td>cntzw</td>
<td>v3.0</td>
<td>C</td>
<td>CTZ Word</td>
<td></td>
<td></td>
</tr>
<tr>
<td>011111 1111111111010</td>
<td>X</td>
<td>99</td>
<td>cntzdn</td>
<td>v3.0</td>
<td>C</td>
<td>CTZ Doubleword</td>
<td></td>
<td></td>
</tr>
<tr>
<td>011111 1111111111010</td>
<td>X</td>
<td>110</td>
<td>snrd</td>
<td>P</td>
<td>SR</td>
<td>Shift Right Algebraic Doubleword</td>
<td></td>
<td></td>
</tr>
<tr>
<td>011111 1111111111011</td>
<td>X</td>
<td>110</td>
<td>srdn</td>
<td>P</td>
<td>SR</td>
<td>Shift Right Algebraic Doubleword Immediate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>011111 1111111111011</td>
<td>X</td>
<td>110</td>
<td>extwz</td>
<td>v3.0</td>
<td>C</td>
<td>Extend Sign Word &amp; Shift Left Immediate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>011111 1111111111010</td>
<td>X</td>
<td>96</td>
<td>extwn</td>
<td>P</td>
<td>SR</td>
<td>Extend Sign Halfword</td>
<td></td>
<td></td>
</tr>
<tr>
<td>011111 1111111111010</td>
<td>X</td>
<td>96</td>
<td>extsn</td>
<td>P</td>
<td>SR</td>
<td>Extend Sign Byte</td>
<td></td>
<td></td>
</tr>
<tr>
<td>011111 1111111111011</td>
<td>X</td>
<td>99</td>
<td>extswz</td>
<td>P</td>
<td>SR</td>
<td>Extend Sign Word</td>
<td></td>
<td></td>
</tr>
<tr>
<td>011111 1111111111010</td>
<td>X</td>
<td>109</td>
<td>addn</td>
<td>P</td>
<td>AND</td>
<td>Add Doubleword</td>
<td></td>
<td></td>
</tr>
<tr>
<td>011111 1111111111011</td>
<td>X</td>
<td>109</td>
<td>sdn</td>
<td>P</td>
<td>SR</td>
<td>AND with Complement</td>
<td></td>
<td></td>
</tr>
<tr>
<td>011111 1111111111011</td>
<td>X</td>
<td>95</td>
<td>andc</td>
<td>P</td>
<td>SR</td>
<td>ANDC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>011111 1111111111011</td>
<td>X</td>
<td>95</td>
<td>nor</td>
<td>P</td>
<td>SR</td>
<td>NOR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>011111 1111111111011</td>
<td>X</td>
<td>100</td>
<td>lpm</td>
<td>v2.0</td>
<td>Bit Permute Doubleword</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011111 1111111111011</td>
<td>X</td>
<td>95</td>
<td>eqv</td>
<td>P</td>
<td>SR</td>
<td>Equivalent</td>
<td></td>
<td></td>
</tr>
<tr>
<td>011111 1111111111011</td>
<td>X</td>
<td>94</td>
<td>xor</td>
<td>P</td>
<td>SR</td>
<td>XOR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>011111 1111111111011</td>
<td>X</td>
<td>95</td>
<td>orc</td>
<td>P</td>
<td>SR</td>
<td>OR with Complement</td>
<td></td>
<td></td>
</tr>
<tr>
<td>011111 1111111111011</td>
<td>X</td>
<td>94</td>
<td>or</td>
<td>P</td>
<td>SR</td>
<td>OR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>011111 1111111111011</td>
<td>X</td>
<td>97</td>
<td>cmpb</td>
<td>v2.0</td>
<td>Compare Byte</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011111 1111111111011</td>
<td>X</td>
<td>876</td>
<td>wait</td>
<td>v3.0</td>
<td>Wait for Interrupt</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010000 1111111111010</td>
<td>D</td>
<td>51</td>
<td>lwz</td>
<td>P</td>
<td>Load Word &amp; Zero</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010000 1111111111010</td>
<td>D</td>
<td>51</td>
<td>wzu</td>
<td>P</td>
<td>Load Word &amp; Zero with Update</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010000 1111111111010</td>
<td>D</td>
<td>48</td>
<td>lbz</td>
<td>P</td>
<td>Load Byte &amp; Zero</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010000 1111111111010</td>
<td>D</td>
<td>48</td>
<td>luzu</td>
<td>P</td>
<td>Load Byte &amp; Zero with Update</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010000 1111111111010</td>
<td>D</td>
<td>46</td>
<td>stw</td>
<td>P</td>
<td>Store Word</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010000 1111111111010</td>
<td>D</td>
<td>56</td>
<td>stwu</td>
<td>P</td>
<td>Store Word with Update</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010000 1111111111010</td>
<td>D</td>
<td>54</td>
<td>stb</td>
<td>P</td>
<td>Store Byte</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010000 1111111111010</td>
<td>D</td>
<td>54</td>
<td>stbu</td>
<td>P</td>
<td>Store Byte with Update</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010000 1111111111010</td>
<td>D</td>
<td>49</td>
<td>lhz</td>
<td>P</td>
<td>Load Halfword &amp; Zero</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010000 1111111111010</td>
<td>D</td>
<td>50</td>
<td>lha</td>
<td>P</td>
<td>Load Halfword Algebraic</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010000 1111111111010</td>
<td>D</td>
<td>50</td>
<td>lhau</td>
<td>P</td>
<td>Load Halfword Algebraic with Update</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010000 1111111111010</td>
<td>D</td>
<td>55</td>
<td>sth</td>
<td>P</td>
<td>Store Halfword</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010000 1111111111010</td>
<td>D</td>
<td>62</td>
<td>tmw</td>
<td>P</td>
<td>Load Multiple Word</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010000 1111111111010</td>
<td>D</td>
<td>62</td>
<td>stmw</td>
<td>P</td>
<td>Store Multiple Word</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010000 1111111111010</td>
<td>D</td>
<td>140</td>
<td>lfs</td>
<td>P</td>
<td>Load Floating Single</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010000 1111111111010</td>
<td>D</td>
<td>141</td>
<td>lfsu</td>
<td>P</td>
<td>Load Floating Single with Update</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010000 1111111111010</td>
<td>D</td>
<td>142</td>
<td>lfd</td>
<td>P</td>
<td>Load Floating Double</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010000 1111111111010</td>
<td>D</td>
<td>142</td>
<td>lfdu</td>
<td>P</td>
<td>Load Floating Double with Update</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010000 1111111111010</td>
<td>D</td>
<td>145</td>
<td>stfs</td>
<td>P</td>
<td>Store Floating Single</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010000 1111111111010</td>
<td>D</td>
<td>145</td>
<td>stfsu</td>
<td>P</td>
<td>Store Floating Single with Update</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010000 1111111111010</td>
<td>D</td>
<td>146</td>
<td>stfd</td>
<td>P</td>
<td>Store Floating Double</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010000 1111111111010</td>
<td>D</td>
<td>146</td>
<td>stfdu</td>
<td>P</td>
<td>Store Floating Double with Update</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010000 1111111111010</td>
<td>D</td>
<td>58</td>
<td>lq</td>
<td>v2.0</td>
<td>Load Quadword</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010000 1111111111010</td>
<td>DS</td>
<td>149</td>
<td>ldpp</td>
<td>v2.0</td>
<td>Load Floating Double Pair</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010000 1111111111010</td>
<td>DS</td>
<td>480</td>
<td>lxsd</td>
<td>v3.0</td>
<td>Load VSX Scalar Doubleword</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010000 1111111111010</td>
<td>DS</td>
<td>485</td>
<td>lxssp</td>
<td>v3.0</td>
<td>Load VSX Scalar Single</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010000 1111111111010</td>
<td>DS</td>
<td>53</td>
<td>ld</td>
<td>P</td>
<td>Load Doubleword</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010000 1111111111010</td>
<td>DS</td>
<td>53</td>
<td>ldu</td>
<td>P</td>
<td>Load Doubleword with Update</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010000 1111111111010</td>
<td>DS</td>
<td>52</td>
<td>lwa</td>
<td>P</td>
<td>Load Word Algebraic</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010000 1111111111010</td>
<td>X</td>
<td>195</td>
<td>dodan</td>
<td>v2.0</td>
<td>DFP Add</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010000 1111111111010</td>
<td>X</td>
<td>195</td>
<td>dmul</td>
<td>v2.0</td>
<td>DFP Multiply</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010000 1111111111010</td>
<td>Z</td>
<td>222</td>
<td>dscil</td>
<td>v2.0</td>
<td>DFP Shift Significand Left Immediate</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010000 1111111111010</td>
<td>Z</td>
<td>222</td>
<td>dscir</td>
<td>v2.0</td>
<td>DFP Shift Significand Right Immediate</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 88. Power ISA AS Instruction Set Sorted by Opcode (Sheet 11 of 18)
**Figure 88. Power ISA AS Instruction Set Sorted by Opcode (Sheet 12 of 18)**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>Book</th>
<th>Page</th>
<th>Mnemonic</th>
<th>Version</th>
<th>Privilege</th>
<th>Mode Dep</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>11001 100000</td>
<td>X</td>
<td>I</td>
<td>199</td>
<td>dcmpo</td>
<td>v2.05</td>
<td>DFP Compare Ordered</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11001 100000</td>
<td>X</td>
<td>I</td>
<td>201</td>
<td>distext</td>
<td>v2.05</td>
<td>DFP Test Exponent</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11111 100000</td>
<td>ZZ2</td>
<td>I</td>
<td>200</td>
<td>distc</td>
<td>v2.05</td>
<td>DFP Test Data Class</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11111 100000</td>
<td>ZZ2</td>
<td>I</td>
<td>200</td>
<td>distd</td>
<td>v2.05</td>
<td>DFP Test Data Group</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11111 100000</td>
<td>X</td>
<td>I</td>
<td>213</td>
<td>dssdp</td>
<td>v2.05</td>
<td>DFP Convert To DFP Long</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11111 100000</td>
<td>X</td>
<td>I</td>
<td>215</td>
<td>dssfl</td>
<td>v2.05</td>
<td>DFP Convert To Fixed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11111 100000</td>
<td>X</td>
<td>I</td>
<td>217</td>
<td>dsslsp</td>
<td>v2.05</td>
<td>DFP Decode DPD To BCD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11111 100000</td>
<td>X</td>
<td>I</td>
<td>218</td>
<td>dixx</td>
<td>v2.05</td>
<td>DFP Extract Exponent</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11111 100000</td>
<td>X</td>
<td>I</td>
<td>193</td>
<td>dsxx</td>
<td>v2.05</td>
<td>DFP Subtract</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11111 100000</td>
<td>X</td>
<td>I</td>
<td>196</td>
<td>ddiv</td>
<td>v2.05</td>
<td>DFP Divide</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11111 100000</td>
<td>X</td>
<td>I</td>
<td>198</td>
<td>dcmpu</td>
<td>v2.05</td>
<td>DFP Compare Unordered</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11111 100000</td>
<td>X</td>
<td>I</td>
<td>202</td>
<td>distfl</td>
<td>v2.05</td>
<td>DFP Test Significance</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11111 100000</td>
<td>X</td>
<td>I</td>
<td>214</td>
<td>dsrfl</td>
<td>v2.05</td>
<td>DFP Round To DFP Short</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11111 100000</td>
<td>X</td>
<td>I</td>
<td>215</td>
<td>dcfll</td>
<td>v2.06</td>
<td>DFP Convert From Fixed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11111 100000</td>
<td>X</td>
<td>I</td>
<td>217</td>
<td>dencdfl</td>
<td>v2.05</td>
<td>DFP Encode BCD To DPD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11111 100000</td>
<td>X</td>
<td>I</td>
<td>218</td>
<td>dixfl</td>
<td>v2.05</td>
<td>DFP Insert Exponent</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11111 100000</td>
<td>000 0011</td>
<td>ZZ3</td>
<td>I</td>
<td>204</td>
<td>dquall</td>
<td>v2.05</td>
<td>DFP Quantize</td>
<td></td>
</tr>
<tr>
<td>11111 100000</td>
<td>001 0011</td>
<td>ZZ3</td>
<td>I</td>
<td>206</td>
<td>drrndl</td>
<td>v2.05</td>
<td>DFP Reround</td>
<td></td>
</tr>
<tr>
<td>11111 100000</td>
<td>010 0011</td>
<td>ZZ3</td>
<td>I</td>
<td>203</td>
<td>dquall</td>
<td>v2.05</td>
<td>DFP Quantize Immediate</td>
<td></td>
</tr>
<tr>
<td>11111 100000</td>
<td>011 0011</td>
<td>ZZ3</td>
<td>I</td>
<td>209</td>
<td>drrndl</td>
<td>v2.05</td>
<td>DFP Round To FP Integer With Inexact</td>
<td></td>
</tr>
<tr>
<td>11111 100000</td>
<td>100 0011</td>
<td>ZZ3</td>
<td>I</td>
<td>211</td>
<td>drrrfl</td>
<td>v2.05</td>
<td>DFP Round To FP Integer Without Inexact</td>
<td></td>
</tr>
<tr>
<td>11111 100000</td>
<td>101 0011</td>
<td>X</td>
<td>I</td>
<td>202</td>
<td>distfl</td>
<td>v3.0</td>
<td>DFP Test Significance Immediate</td>
<td></td>
</tr>
<tr>
<td>11111 100010</td>
<td>1010 0110</td>
<td>X</td>
<td>I</td>
<td>164</td>
<td>fclflsl</td>
<td>v2.06</td>
<td>Floating Convert with round Signed Doubleword to Single-Precision format</td>
<td></td>
</tr>
<tr>
<td>11111 100010</td>
<td>1011 0110</td>
<td>X</td>
<td>I</td>
<td>165</td>
<td>fclflus</td>
<td>v2.06</td>
<td>Floating Convert with round Unsigned Doubleword to Single-Precision format</td>
<td></td>
</tr>
<tr>
<td>11111 100100</td>
<td>1010 0110</td>
<td>A</td>
<td>I</td>
<td>153</td>
<td>fdivslv</td>
<td>PPC</td>
<td>Floating Divide Single</td>
<td></td>
</tr>
<tr>
<td>11111 100100</td>
<td>1011 0110</td>
<td>A</td>
<td>I</td>
<td>152</td>
<td>faddslv</td>
<td>PPC</td>
<td>Floating Add Single</td>
<td></td>
</tr>
<tr>
<td>11111 100100</td>
<td>1100 0110</td>
<td>A</td>
<td>I</td>
<td>154</td>
<td>fsgtalv</td>
<td>PPC</td>
<td>Floating Square Root Single</td>
<td></td>
</tr>
<tr>
<td>11111 100100</td>
<td>1101 0110</td>
<td>A</td>
<td>I</td>
<td>154</td>
<td>freislv</td>
<td>PPC</td>
<td>Floating Reciprocal Estimate Single</td>
<td></td>
</tr>
<tr>
<td>11111 101000</td>
<td>1110 0110</td>
<td>A</td>
<td>I</td>
<td>155</td>
<td>fmulsav</td>
<td>PPC</td>
<td>Floating Multiply Single</td>
<td></td>
</tr>
<tr>
<td>11111 101000</td>
<td>1111 0110</td>
<td>A</td>
<td>I</td>
<td>155</td>
<td>frsrtslv</td>
<td>v2.02</td>
<td>Floating Reciprocal Square Root Estimate Single</td>
<td></td>
</tr>
<tr>
<td>11111 101000</td>
<td>1110 0110</td>
<td>A</td>
<td>I</td>
<td>158</td>
<td>fsubslv</td>
<td>PPC</td>
<td>Floating Subtract Single</td>
<td></td>
</tr>
<tr>
<td>11111 101000</td>
<td>1111 0110</td>
<td>A</td>
<td>I</td>
<td>157</td>
<td>fmaddslv</td>
<td>PPC</td>
<td>Floating Multiply-Add Single</td>
<td></td>
</tr>
<tr>
<td>11111 101000</td>
<td>1110 0110</td>
<td>A</td>
<td>I</td>
<td>158</td>
<td>fnmsubslv</td>
<td>PPC</td>
<td>Floating Negative Multiply-Subtract Single</td>
<td></td>
</tr>
<tr>
<td>11111 101000</td>
<td>1111 0110</td>
<td>A</td>
<td>I</td>
<td>158</td>
<td>fnmaddsllv</td>
<td>PPC</td>
<td>Floating Negative Multiply-Add Single</td>
<td></td>
</tr>
<tr>
<td>11110 000000</td>
<td>0000 0000</td>
<td>XX3</td>
<td>I</td>
<td>518</td>
<td>xasaddsp</td>
<td>v2.07</td>
<td>VSX Scalar Add Single-Precision</td>
<td></td>
</tr>
<tr>
<td>11110 000000</td>
<td>0001 0000</td>
<td>XX3</td>
<td>I</td>
<td>649</td>
<td>xssubsp</td>
<td>v2.07</td>
<td>VSX Scalar Subtract Single-Precision</td>
<td></td>
</tr>
<tr>
<td>11110 000000</td>
<td>0002 0000</td>
<td>XX3</td>
<td>I</td>
<td>604</td>
<td>xmulsp</td>
<td>v2.07</td>
<td>VSX Scalar Multiply Single-Precision</td>
<td></td>
</tr>
<tr>
<td>11110 000000</td>
<td>0003 0000</td>
<td>XX3</td>
<td>I</td>
<td>566</td>
<td>xsdivsp</td>
<td>v2.07</td>
<td>VSX Scalar Divide Single-Precision</td>
<td></td>
</tr>
<tr>
<td>11110 000000</td>
<td>0010 0000</td>
<td>XX3</td>
<td>I</td>
<td>513</td>
<td>xasadddp</td>
<td>v2.06</td>
<td>VSX Scalar Add Double-Precision</td>
<td></td>
</tr>
<tr>
<td>11110 000000</td>
<td>0011 0000</td>
<td>XX3</td>
<td>I</td>
<td>645</td>
<td>xssubdp</td>
<td>v2.06</td>
<td>VSX Scalar Subtract Double-Precision</td>
<td></td>
</tr>
<tr>
<td>11110 000000</td>
<td>0010 0000</td>
<td>XX3</td>
<td>I</td>
<td>600</td>
<td>xmuldp</td>
<td>v2.06</td>
<td>VSX Scalar Multiply Double-Precision</td>
<td></td>
</tr>
<tr>
<td>11110 000000</td>
<td>0011 0000</td>
<td>XX3</td>
<td>I</td>
<td>562</td>
<td>xdivdp</td>
<td>v2.06</td>
<td>VSX Scalar Divide Double-Precision</td>
<td></td>
</tr>
<tr>
<td>11110 010000</td>
<td>0100 0000</td>
<td>XX3</td>
<td>I</td>
<td>663</td>
<td>xasaddsp</td>
<td>v2.06</td>
<td>VSX Vector Add Single-Precision</td>
<td></td>
</tr>
<tr>
<td>11110 010000</td>
<td>0101 0000</td>
<td>XX3</td>
<td>I</td>
<td>755</td>
<td>xsubsp</td>
<td>v2.06</td>
<td>VSX Vector Subtract Single-Precision</td>
<td></td>
</tr>
<tr>
<td>11110 010000</td>
<td>0110 0000</td>
<td>XX3</td>
<td>I</td>
<td>723</td>
<td>xmulsp</td>
<td>v2.06</td>
<td>VSX Vector Multiply Single-Precision</td>
<td></td>
</tr>
<tr>
<td>11110 010000</td>
<td>0111 0000</td>
<td>XX3</td>
<td>I</td>
<td>698</td>
<td>xdivsp</td>
<td>v2.06</td>
<td>VSX Vector Divide Single-Precision</td>
<td></td>
</tr>
<tr>
<td>11110 010000</td>
<td>1000 0000</td>
<td>XX3</td>
<td>I</td>
<td>659</td>
<td>xasadddp</td>
<td>v2.06</td>
<td>VSX Vector Add Double-Precision</td>
<td></td>
</tr>
<tr>
<td>11110 010000</td>
<td>1001 0000</td>
<td>XX3</td>
<td>I</td>
<td>753</td>
<td>xsubdp</td>
<td>v2.06</td>
<td>VSX Vector Subtract Double-Precision</td>
<td></td>
</tr>
<tr>
<td>11110 010000</td>
<td>1010 0000</td>
<td>XX3</td>
<td>I</td>
<td>721</td>
<td>xmuldp</td>
<td>v2.06</td>
<td>VSX Vector Multiply Double-Precision</td>
<td></td>
</tr>
<tr>
<td>11110 010000</td>
<td>1011 0000</td>
<td>XX3</td>
<td>I</td>
<td>696</td>
<td>xdivdp</td>
<td>v2.06</td>
<td>VSX Vector Divide Double-Precision</td>
<td></td>
</tr>
<tr>
<td>11110 100000</td>
<td>1000 0000</td>
<td>XX3</td>
<td>I</td>
<td>581</td>
<td>xmacdp</td>
<td>v3.0</td>
<td>VSX Scalar Maximum Type-C Double-Precision</td>
<td></td>
</tr>
<tr>
<td>11110 100000</td>
<td>1001 0000</td>
<td>XX3</td>
<td>I</td>
<td>587</td>
<td>xmincdp</td>
<td>v3.0</td>
<td>VSX Scalar Minimum Type-C Double-Precision</td>
<td></td>
</tr>
<tr>
<td>11110 100000</td>
<td>1010 0000</td>
<td>XX3</td>
<td>I</td>
<td>583</td>
<td>xmacdp</td>
<td>v3.0</td>
<td>VSX Scalar Maximum Type-J Double-Precision</td>
<td></td>
</tr>
<tr>
<td>11110 100000</td>
<td>1011 0000</td>
<td>XX3</td>
<td>I</td>
<td>589</td>
<td>xmincdp</td>
<td>v3.0</td>
<td>VSX Scalar Minimum Type-J Double-Precision</td>
<td></td>
</tr>
<tr>
<td>Instruction¹</td>
<td>Format</td>
<td>Book</td>
<td>Page</td>
<td>Mnemonic</td>
<td>Version²</td>
<td>Privilege³</td>
<td>Mode Dep⁴</td>
<td>Name</td>
</tr>
<tr>
<td>--------------</td>
<td>--------</td>
<td>------</td>
<td>------</td>
<td>----------</td>
<td>----------</td>
<td>------------</td>
<td>-----------</td>
<td>------</td>
</tr>
<tr>
<td>0000000000</td>
<td>XX3</td>
<td>579</td>
<td>v2.06</td>
<td>xsmmaxp</td>
<td>VSX Scalar Maximum Double-Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000000010</td>
<td>XX3</td>
<td>585</td>
<td>v2.06</td>
<td>xsmindp</td>
<td>VSX Scalar Minimum Double-Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000001000</td>
<td>XX3</td>
<td>533</td>
<td>v2.06</td>
<td>xscpsgrdp</td>
<td>VSX Scalar Copy Sign Double-Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000001010</td>
<td>XX3</td>
<td>709</td>
<td>v2.06</td>
<td>xvmmaxp</td>
<td>VSX Vector Maximum Single-Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000001010</td>
<td>XX3</td>
<td>713</td>
<td>v2.06</td>
<td>xvmmsp</td>
<td>VSX Vector Minimum Single-Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000010000</td>
<td>XX3</td>
<td>671</td>
<td>v2.06</td>
<td>xcpsgrnp</td>
<td>VSX Vector Copy Sign Single-Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000010001</td>
<td>XX1</td>
<td>700</td>
<td>v3.0</td>
<td>xvexpsp</td>
<td>VSX Vector Insert Exponent Single-Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000010010</td>
<td>XX3</td>
<td>707</td>
<td>v2.06</td>
<td>xvmaxp</td>
<td>VSX Vector Maximum Double-Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000010010</td>
<td>XX3</td>
<td>711</td>
<td>v2.06</td>
<td>xvmindp</td>
<td>VSX Vector Minimum Double-Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000010010</td>
<td>XX3</td>
<td>671</td>
<td>v2.06</td>
<td>xcpsgrdp</td>
<td>VSX Vector Copy Sign Double-Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000010010</td>
<td>XX3</td>
<td>700</td>
<td>v3.0</td>
<td>xvexpsp</td>
<td>VSX Vector Insert Exponent Double-Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000010010</td>
<td>XX3</td>
<td>573</td>
<td>v2.07</td>
<td>xsmaddasp</td>
<td>VSX Scalar Multiply-Add Type-A Single-Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000010010</td>
<td>XX3</td>
<td>573</td>
<td>v2.07</td>
<td>xsmaddmsp</td>
<td>VSX Scalar Multiply-Add Type-M Single-Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000010100</td>
<td>XX3</td>
<td>594</td>
<td>v2.07</td>
<td>xsmsubasp</td>
<td>VSX Scalar Multiply-Subtract Type-A Single-Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000010101</td>
<td>XX3</td>
<td>594</td>
<td>v2.07</td>
<td>xsmsubmsp</td>
<td>VSX Scalar Multiply-Subtract Type-M Single-Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000010110</td>
<td>XX3</td>
<td>570</td>
<td>v2.06</td>
<td>xsmaddadp</td>
<td>VSX Scalar Multiply-Add Type-A Double-Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000010110</td>
<td>XX3</td>
<td>570</td>
<td>v2.06</td>
<td>xsmaddmdp</td>
<td>VSX Scalar Multiply-Add Type-M Double-Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000010110</td>
<td>XX3</td>
<td>591</td>
<td>v2.06</td>
<td>xsmsubadp</td>
<td>VSX Scalar Multiply-Subtract Type-A Double-Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000010110</td>
<td>XX3</td>
<td>591</td>
<td>v2.06</td>
<td>xsmsubmdp</td>
<td>VSX Scalar Multiply-Subtract Type-M Double-Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000010110</td>
<td>XX3</td>
<td>704</td>
<td>v2.06</td>
<td>xvmaddasp</td>
<td>VSX Vector Multiply-Add Type-A Single-Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000010110</td>
<td>XX3</td>
<td>704</td>
<td>v2.06</td>
<td>xvmaddmsp</td>
<td>VSX Vector Multiply-Add Type-M Single-Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000010110</td>
<td>XX3</td>
<td>716</td>
<td>v2.06</td>
<td>xvmsubasp</td>
<td>VSX Vector Multiply-Subtract Type-A Single-Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000010110</td>
<td>XX3</td>
<td>716</td>
<td>v2.06</td>
<td>xvmsubmsp</td>
<td>VSX Vector Multiply-Subtract Type-M Single-Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000010111</td>
<td>XX3</td>
<td>701</td>
<td>v2.06</td>
<td>xvmaddadp</td>
<td>VSX Vector Multiply-Add Type-A Double-Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000010111</td>
<td>XX3</td>
<td>701</td>
<td>v2.06</td>
<td>xvmaddmdp</td>
<td>VSX Vector Multiply-Add Type-M Double-Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000011000</td>
<td>XX3</td>
<td>613</td>
<td>v2.07</td>
<td>xsnmaddasp</td>
<td>VSX Scalar Negative Multiply-Add Type-A Single-Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000011000</td>
<td>XX3</td>
<td>613</td>
<td>v2.07</td>
<td>xsnmaddmsp</td>
<td>VSX Scalar Negative Multiply-Add Type-M Single-Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000011001</td>
<td>XX3</td>
<td>622</td>
<td>v2.07</td>
<td>xsnsubasp</td>
<td>VSX Scalar Negative Multiply-Subtract Type-A Single-Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000011001</td>
<td>XX3</td>
<td>622</td>
<td>v2.07</td>
<td>xsnsubmsp</td>
<td>VSX Scalar Negative Multiply-Subtract Type-M Single-Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000011010</td>
<td>XX3</td>
<td>606</td>
<td>v2.06</td>
<td>xsnmaddadp</td>
<td>VSX Scalar Negative Multiply-Add Type-A Double-Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000011010</td>
<td>XX3</td>
<td>606</td>
<td>v2.06</td>
<td>xsnmaddmdp</td>
<td>VSX Scalar Negative Multiply-Add Type-M Double-Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000011010</td>
<td>XX3</td>
<td>619</td>
<td>v2.06</td>
<td>xsnsubadp</td>
<td>VSX Scalar Negative Multiply-Subtract Type-A Double-Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000011010</td>
<td>XX3</td>
<td>619</td>
<td>v2.06</td>
<td>xsnsubmdp</td>
<td>VSX Scalar Negative Multiply-Subtract Type-M Double-Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000011010</td>
<td>XX3</td>
<td>732</td>
<td>v2.06</td>
<td>xvnmaddasp</td>
<td>VSX Vector Negative Multiply-Add Type-A Single-Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000011010</td>
<td>XX3</td>
<td>732</td>
<td>v2.06</td>
<td>xvnmaddmsp</td>
<td>VSX Vector Negative Multiply-Add Type-M Single-Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000011010</td>
<td>XX3</td>
<td>736</td>
<td>v2.06</td>
<td>xvnsubasp</td>
<td>VSX Vector Negative Multiply-Subtract Type-A Single-Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000011010</td>
<td>XX3</td>
<td>736</td>
<td>v2.06</td>
<td>xvnsubmsp</td>
<td>VSX Vector Negative Multiply-Subtract Type-M Single-Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000011011</td>
<td>XX3</td>
<td>727</td>
<td>v2.06</td>
<td>xvnmaddadp</td>
<td>VSX Vector Negative Multiply-Add Type-A Double-Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000011011</td>
<td>XX3</td>
<td>727</td>
<td>v2.06</td>
<td>xvnmaddmdp</td>
<td>VSX Vector Negative Multiply-Add Type-M Double-Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000011011</td>
<td>XX3</td>
<td>735</td>
<td>v2.06</td>
<td>xvnsubadp</td>
<td>VSX Vector Negative Multiply-Subtract Type-A Double-Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000011011</td>
<td>XX3</td>
<td>735</td>
<td>v2.06</td>
<td>xvnsubmdp</td>
<td>VSX Vector Negative Multiply-Subtract Type-M Double-Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000011100</td>
<td>XX3</td>
<td>774</td>
<td>v2.06</td>
<td>xoxsidwi</td>
<td>VSX Vector Shift Left Double by Word Immediate</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000011100</td>
<td>XX3</td>
<td>773</td>
<td>v2.06</td>
<td>xopermdd</td>
<td>VSX Vector Doubleword Permute Immediate</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000011101</td>
<td>XX3</td>
<td>771</td>
<td>v2.06</td>
<td>xoxmgwh</td>
<td>VSX Vector Merge Word High</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000011110</td>
<td>XX3</td>
<td>772</td>
<td>v3.0</td>
<td>xopernm</td>
<td>VSX Vector Permute</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000011110</td>
<td>XX3</td>
<td>771</td>
<td>v2.06</td>
<td>xoxmgwv</td>
<td>VSX Vector Merge Word Low</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000011110</td>
<td>XX3</td>
<td>772</td>
<td>v3.0</td>
<td>xopemrr</td>
<td>VSX Vector Permute Right-indexed</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000011110</td>
<td>XX3</td>
<td>774</td>
<td>v2.06</td>
<td>xospitw</td>
<td>VSX Vector Split Word</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000011110</td>
<td>XX3</td>
<td>714</td>
<td>v2.06</td>
<td>xoxpsib</td>
<td>VSX Vector Split Immediate Byte</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000011110</td>
<td>XX3</td>
<td>761</td>
<td>v2.06</td>
<td>xoland</td>
<td>VSX Vector Logical AND</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000011110</td>
<td>XX3</td>
<td>767</td>
<td>v2.06</td>
<td>xolandc</td>
<td>VSX Vector Logical AND with Complement</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000011110</td>
<td>XX3</td>
<td>770</td>
<td>v2.06</td>
<td>xxloxx</td>
<td>VSX Vector Logical OR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000011110</td>
<td>XX3</td>
<td>770</td>
<td>v2.06</td>
<td>xxloxx</td>
<td>VSX Vector Logical XOR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000011110</td>
<td>XX3</td>
<td>769</td>
<td>v2.06</td>
<td>xxloxx</td>
<td>VSX Vector Logical NOR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction¹</td>
<td>Format</td>
<td>Book</td>
<td>Page</td>
<td>Mnemonic</td>
<td>Version²</td>
<td>Privilege³</td>
<td>Mode Dep⁴</td>
<td>Name</td>
</tr>
<tr>
<td>---------------</td>
<td>--------</td>
<td>------</td>
<td>------</td>
<td>----------</td>
<td>----------</td>
<td>------------</td>
<td>----------</td>
<td>------</td>
</tr>
<tr>
<td>111100 ....... 10101 030...</td>
<td>XX3</td>
<td>I</td>
<td>769</td>
<td>xxlorc</td>
<td>v2.07</td>
<td>VSX Vector Logical OR with Complement</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111100 ....... 10110 030...</td>
<td>XX3</td>
<td>I</td>
<td>768</td>
<td>xxinand</td>
<td>v2.07</td>
<td>VSX Vector Logical NAND</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111100 ....... 10111 030...</td>
<td>XX3</td>
<td>I</td>
<td>768</td>
<td>xxleqv</td>
<td>v2.07</td>
<td>VSX Vector Logical Equivalence</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111100 ....... 01100 031...</td>
<td>XX2</td>
<td>I</td>
<td>766</td>
<td>xxxextractuw</td>
<td>v3.0</td>
<td>VSX Vector Extract Unsigned Word</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111100 ....... 01101 031...</td>
<td>XX2</td>
<td>I</td>
<td>766</td>
<td>xxxint2uw</td>
<td>v3.0</td>
<td>VSX Vector Insert Word</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111100 ....... 00000 011...</td>
<td>XX3</td>
<td>I</td>
<td>524</td>
<td>xxxcmpeqdp</td>
<td>v3.0</td>
<td>VSX Scalar Compare Equal Double-Precision</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111100 ....... 00010 011...</td>
<td>XX3</td>
<td>I</td>
<td>526</td>
<td>xxxcmpgtdp</td>
<td>v3.0</td>
<td>VSX Scalar Compare Greater Than Double-Precision</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111100 ....... 00100 011...</td>
<td>XX3</td>
<td>I</td>
<td>525</td>
<td>xxxcmpeqdp</td>
<td>v4.0</td>
<td>VSX Scalar Compare Equal Than or Equal Double-Precision</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111100 ....... 00101 011...</td>
<td>XX3</td>
<td>I</td>
<td>532</td>
<td>xxxcmpuqdp</td>
<td>v2.06</td>
<td>VSX Scalar Compare Unordered Double-Precision</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111100 ....... 00110 011...</td>
<td>XX3</td>
<td>I</td>
<td>527</td>
<td>xxxcmpodp</td>
<td>v2.06</td>
<td>VSX Scalar Compare Ordered Double-Precision</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111100 ....... 00111 011...</td>
<td>XX3</td>
<td>I</td>
<td>522</td>
<td>xxxcmpeqdp</td>
<td>v4.0</td>
<td>VSX Scalar Compare Exponents Double-Precision</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111100 ....... 10000 010...</td>
<td>XX3</td>
<td>I</td>
<td>666</td>
<td>xvcmpexpdp</td>
<td>v2.06</td>
<td>VSX Vector Compare Equal Single-Precision</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111100 ....... 10001 010...</td>
<td>XX3</td>
<td>I</td>
<td>670</td>
<td>xvcmpgtsp</td>
<td>v2.06</td>
<td>VSX Vector Compare Greater Than Single-Precision</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111100 ....... 10010 010...</td>
<td>XX3</td>
<td>I</td>
<td>668</td>
<td>xvcmpinge</td>
<td>v2.06</td>
<td>VSX Vector Compare Greater Than or Equal Single-Precision</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111100 ....... 10011 010...</td>
<td>XX3</td>
<td>I</td>
<td>665</td>
<td>xvcmpune</td>
<td>v2.06</td>
<td>VSX Vector Compare Equal Double-Precision</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111100 ....... 10100 010...</td>
<td>XX3</td>
<td>I</td>
<td>669</td>
<td>xvcmpgtsp</td>
<td>v2.06</td>
<td>VSX Vector Compare Greater Than Double-Precision</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111100 ....... 10101 010...</td>
<td>XX3</td>
<td>I</td>
<td>675</td>
<td>xvcmpug</td>
<td>v2.06</td>
<td>VSX Vector Compare Double-Precision</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111100 ....... 10110 010...</td>
<td>XX3</td>
<td>I</td>
<td>669</td>
<td>xvcmpudp</td>
<td>v2.06</td>
<td>VSX Vector Compare Greater Than Equal Double-Precision</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111100 ....... 10111 010...</td>
<td>XX3</td>
<td>I</td>
<td>673</td>
<td>xvcmpup</td>
<td>v2.06</td>
<td>VSX Vector Convert with round to zero Double-Precision to Signed Word format</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 88. Power ISA AS Instruction Set Sorted by Opcode (Sheet 14 of 18)

1192 Power ISA™ Appendices
<table>
<thead>
<tr>
<th>Instruction^1</th>
<th>Format</th>
<th>Book</th>
<th>Page</th>
<th>Mnemonic</th>
<th>Version^2</th>
<th>Privilege^3</th>
<th>Mode Dep^4</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>111100 ...... /// /// ...... 01110 1011..</td>
<td>XX2</td>
<td>I</td>
<td>692</td>
<td>vcvcxsddp</td>
<td>v2.06</td>
<td>VSX Vector Convert with round Signed Doubleword to Double-Precision format</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111100 ...... /// /// ...... 01100 1011..</td>
<td>XX2</td>
<td>I</td>
<td>631</td>
<td>xsrddi</td>
<td>v2.06</td>
<td>VSX Scalar Round Double-Precision to Integral</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111100 ...... /// /// ...... 01010 1011..</td>
<td>XX2</td>
<td>I</td>
<td>630</td>
<td>xsrddp</td>
<td>v2.06</td>
<td>VSX Scalar Round Double-Precision to Integral toward Zero</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111100 ...... /// /// ...... 01000 1011..</td>
<td>XX2</td>
<td>I</td>
<td>746</td>
<td>xsvsip</td>
<td>v2.06</td>
<td>VSX Vector Round Single-Precision to Integral toward +Infinity</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111100 ...... /// /// ...... 00110 1011..</td>
<td>XX2</td>
<td>I</td>
<td>747</td>
<td>xsvspiz</td>
<td>v2.06</td>
<td>VSX Vector Round Single-Precision to Integral toward -Infinity</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111100 ...... /// /// ...... 00100 1011..</td>
<td>XX2</td>
<td>I</td>
<td>741</td>
<td>xsvdpi</td>
<td>v2.06</td>
<td>VSX Vector Round Double-Precision to Integral</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111100 ...... /// /// ...... 00010 1011..</td>
<td>XX2</td>
<td>I</td>
<td>742</td>
<td>xsvdpi</td>
<td>v2.06</td>
<td>VSX Vector Round Double-Precision to Integral toward +Infinity</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111100 ...... /// /// ...... 00000 1011..</td>
<td>XX2</td>
<td>I</td>
<td>743</td>
<td>xsvdpiz</td>
<td>v2.06</td>
<td>VSX Vector Round Double-Precision to Integral toward -Infinity</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111100 ...... /// /// ...... 00001 1011..</td>
<td>XX2</td>
<td>I</td>
<td>536</td>
<td>xsqvdpdp</td>
<td>v2.06</td>
<td>VSX Scalar Convert with round Double-Precision to Single-Precision format</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111100 ...... /// /// ...... 01001 1011..</td>
<td>XX2</td>
<td>I</td>
<td>638</td>
<td>xsvsp</td>
<td>v2.07</td>
<td>VSX Scalar Round Double-Precision to Single-Precision</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111100 ...... /// /// ...... 01111 101../</td>
<td>XX2</td>
<td>I</td>
<td>651</td>
<td>xsvsqrtdp</td>
<td>v2.06</td>
<td>VSX Scalar Convert Single-Precision to Double-Precision format</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111100 ...... /// /// ...... 10110 100./</td>
<td>XX2</td>
<td>I</td>
<td>512</td>
<td>xsabsdp</td>
<td>v2.06</td>
<td>VSX Scalar Absolute Double-Precision</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111100 ...... /// /// ...... 10100 100..</td>
<td>XX2</td>
<td>I</td>
<td>606</td>
<td>xsabsdp</td>
<td>v2.06</td>
<td>VSX Scalar Absolute Double-Precision</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111100 ...... /// /// ...... 10010 100..</td>
<td>XX2</td>
<td>I</td>
<td>607</td>
<td>xsnegdps</td>
<td>v2.06</td>
<td>VSX Scalar Negate Double-Precision</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111100 ...... /// /// ...... 10000 100..</td>
<td>XX2</td>
<td>I</td>
<td>672</td>
<td>xsdvsp</td>
<td>v2.06</td>
<td>VSX Vector Convert with round Double-Precision to Single-Precision format</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111100 ...... /// /// ...... 11000 100..</td>
<td>XX2</td>
<td>I</td>
<td>725</td>
<td>xmvabsssp</td>
<td>v2.06</td>
<td>VSX Vector Negative Absolute Single-Precision</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111100 ...... /// /// ...... 11111 100..</td>
<td>XX2</td>
<td>I</td>
<td>726</td>
<td>xmvepsp</td>
<td>v2.06</td>
<td>VSX Vector Negate Single-Precision</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111100 ...... /// /// ...... 10100 100..</td>
<td>XX2</td>
<td>I</td>
<td>682</td>
<td>xsvvsdpdp</td>
<td>v2.06</td>
<td>VSX Scalar Convert Single-Precision to Double-Precision format</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111100 ...... /// /// ...... 11010 100..</td>
<td>XX2</td>
<td>I</td>
<td>656</td>
<td>xmvabsdp</td>
<td>v2.06</td>
<td>VSX Vector Absolute Double-Precision</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111100 ...... /// /// ...... 11100 100..</td>
<td>XX2</td>
<td>I</td>
<td>725</td>
<td>xmvabsdp</td>
<td>v2.06</td>
<td>VSX Vector Absolute Double-Precision</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111100 ...... /// /// ...... 11101 100..</td>
<td>XX2</td>
<td>I</td>
<td>726</td>
<td>xmvepdp</td>
<td>v2.06</td>
<td>VSX Vector Negate Double-Precision</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111100 ...... /// /// ...... 11111 100..</td>
<td>XX2</td>
<td>I</td>
<td>640</td>
<td>xrsqrepstsp</td>
<td>v2.07</td>
<td>VSX Scalar Reciprocal Square Root Estimate Single-Precision</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111100 ...... /// /// ...... 00110 100..</td>
<td>XX2</td>
<td>I</td>
<td>633</td>
<td>xrsesp</td>
<td>v2.07</td>
<td>VSX Scalar Reciprocal Estimate Single-Precision</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111100 ...... /// /// ...... 01010 100..</td>
<td>XX2</td>
<td>I</td>
<td>639</td>
<td>xrsqrtcedp</td>
<td>v2.06</td>
<td>VSX Scalar Reciprocal Square Root Estimate Double-Precision</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111100 ...... /// /// ...... 01001 100..</td>
<td>XX2</td>
<td>I</td>
<td>632</td>
<td>xrsrip</td>
<td>v2.06</td>
<td>VSX Scalar Reciprocal Estimate Double-Precision</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111100 ...... /// /// ...... 01111 100..</td>
<td>XX2</td>
<td>I</td>
<td>652</td>
<td>xsrstgdpp</td>
<td>v2.06</td>
<td>VSX Scalar Test for software Square Root Double-Precision</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111100 ...... /// /// ...... 10110 100..</td>
<td>XX2</td>
<td>I</td>
<td>651</td>
<td>xsxtdvdp</td>
<td>v2.06</td>
<td>VSX Scalar Test for software Divide Double-Precision</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111100 ...... /// /// ...... 11000 100..</td>
<td>XX2</td>
<td>I</td>
<td>750</td>
<td>xsvrsqresp</td>
<td>v2.06</td>
<td>VSX Vector Reciprocal Square Root Estimate Single-Precision</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111100 ...... /// /// ...... 11010 100..</td>
<td>XX2</td>
<td>I</td>
<td>745</td>
<td>xsvresp</td>
<td>v2.06</td>
<td>VSX Vector Reciprocal Estimate Single-Precision</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111100 ...... /// /// ...... 11011 100..</td>
<td>XX2</td>
<td>I</td>
<td>759</td>
<td>xsvrsqisp</td>
<td>v2.06</td>
<td>VSX Vector Test for software Square Root Single-Precision</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111100 ...... /// /// ...... 11110 100..</td>
<td>XX2</td>
<td>I</td>
<td>758</td>
<td>xsvrdisp</td>
<td>v2.06</td>
<td>VSX Vector Test for software Divide Double-Precision</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111100 ...... /// /// ...... 11111 100..</td>
<td>XX2</td>
<td>I</td>
<td>748</td>
<td>xsvrdtpdp</td>
<td>v2.06</td>
<td>VSX Vector Reciprocal Square Root Estimate Double-Precision</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111100 ...... /// /// ...... 10111 100..</td>
<td>XX2</td>
<td>I</td>
<td>744</td>
<td>xsvrdp</td>
<td>v2.06</td>
<td>VSX Vector Reciprocal Estimate Double-Precision</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111100 ...... /// /// ...... 11101 100..</td>
<td>XX2</td>
<td>I</td>
<td>759</td>
<td>xsvrsfdtp</td>
<td>v2.06</td>
<td>VSX Vector Test for software Square Root Double-Precision</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111100 ...... /// /// ...... 11110 100..</td>
<td>XX2</td>
<td>I</td>
<td>751</td>
<td>xsvrdvp</td>
<td>v2.06</td>
<td>VSX Vector Test for software Divide Double-Precision</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111100 ...... /// /// ...... 11111 100..</td>
<td>XX2</td>
<td>I</td>
<td>655</td>
<td>xsvsddscsp</td>
<td>v3.0</td>
<td>VSX Scalar Test Data Class Single-Precision</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111100 ...... /// /// ...... 11110 100..</td>
<td>XX2</td>
<td>I</td>
<td>655</td>
<td>xsvsddcdp</td>
<td>v3.0</td>
<td>VSX Scalar Test Data Class Double-Precision</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111100 ...... /// /// ...... 11101 100..</td>
<td>XX2</td>
<td>I</td>
<td>761</td>
<td>xsvsddcdp</td>
<td>v3.0</td>
<td>VSX Vector Test Data Class Single-Precision</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111100 ...... /// /// ...... 11100 100..</td>
<td>XX2</td>
<td>I</td>
<td>760</td>
<td>xsvsddcdp</td>
<td>v3.0</td>
<td>VSX Vector Test Data Class Double-Precision</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111100 ...... /// /// ...... 11000 100..</td>
<td>XX2</td>
<td>I</td>
<td>644</td>
<td>xsvsqsp</td>
<td>v2.07</td>
<td>VSX Scalar Square Root Single-Precision</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111100 ...... /// /// ...... 10100 100..</td>
<td>XX2</td>
<td>I</td>
<td>641</td>
<td>xsvsqstsp</td>
<td>v2.06</td>
<td>VSX Scalar Square Root Double-Precision</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111100 ...... /// /// ...... 10010 100..</td>
<td>XX2</td>
<td>I</td>
<td>629</td>
<td>xsvrdsic</td>
<td>v2.06</td>
<td>VSX Scalar Round Double-Precision to Integral using Current rounding mode</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111100 ...... /// /// ...... 10000 100..</td>
<td>XX2</td>
<td>I</td>
<td>752</td>
<td>xsvstqsp</td>
<td>v2.06</td>
<td>VSX Vector Square Root Single-Precision</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111100 ...... /// /// ...... 01110 100..</td>
<td>XX2</td>
<td>I</td>
<td>746</td>
<td>xsvrsic</td>
<td>v2.06</td>
<td>VSX Vector Square Root Single-Precision to Integral using Current rounding mode</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111100 ...... /// /// ...... 01100 100..</td>
<td>XX2</td>
<td>I</td>
<td>751</td>
<td>xsvstqdp</td>
<td>v2.06</td>
<td>VSX Vector Square Root Double-Precision</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111100 ...... /// /// ...... 01010 100..</td>
<td>XX2</td>
<td>I</td>
<td>741</td>
<td>xsvrdp</td>
<td>v2.06</td>
<td>VSX Vector Round Double-Precision to Integral using Current rounding mode</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 88. Power ISA AS Instruction Set Sorted by Opcode (Sheet 15 of 18)
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>Book</th>
<th>Page</th>
<th>Mnemonic</th>
<th>Version</th>
<th>Privilege</th>
<th>Mode Dep.</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>111100</td>
<td>111111</td>
<td>X2</td>
<td>1</td>
<td>xscvdpspn</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>VSX Scalar Convert Double-Precision to Single-Precision Non-signalling format</td>
</tr>
<tr>
<td>111100</td>
<td>111111</td>
<td>X2</td>
<td>1</td>
<td>xscvdppdn</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>VSX Scalar Convert Single-Precision to Double-Precision Non-signalling format</td>
</tr>
<tr>
<td>111100</td>
<td>111000</td>
<td>X2</td>
<td>1</td>
<td>xxsexdp</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>VSX Scalar Extract Exponent Double-Precision</td>
</tr>
<tr>
<td>111100</td>
<td>111000</td>
<td>X2</td>
<td>1</td>
<td>xxsexgdp</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>VSX Scalar Extract Significant Double-Precision</td>
</tr>
<tr>
<td>111100</td>
<td>111000</td>
<td>X2</td>
<td>1</td>
<td>xsbrr</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>VSX Vector Byte-Reverse Halfword</td>
</tr>
<tr>
<td>111100</td>
<td>111000</td>
<td>X2</td>
<td>1</td>
<td>xxsexsp</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>VSX Vector Extract Exponent Single-Precision</td>
</tr>
<tr>
<td>111100</td>
<td>111000</td>
<td>X2</td>
<td>1</td>
<td>xxsexgp</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>VSX Vector Extract Significant Single-Precision</td>
</tr>
<tr>
<td>111100</td>
<td>111000</td>
<td>X2</td>
<td>1</td>
<td>xdbufv</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>VSX Vector Byte-Reverse Word</td>
</tr>
<tr>
<td>111100</td>
<td>111000</td>
<td>X2</td>
<td>1</td>
<td>xxbrd</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>VSX Vector Byte-Reverse Doubleword</td>
</tr>
<tr>
<td>111100</td>
<td>111000</td>
<td>X2</td>
<td>1</td>
<td>xvcvvp</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>VSX Vector Convert Half-Precision to Single-Precision format</td>
</tr>
<tr>
<td>111100</td>
<td>111000</td>
<td>X2</td>
<td>1</td>
<td>xvcvshp</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>VSX Vector Convert with round Single-Precision to Half-Precision format</td>
</tr>
<tr>
<td>111100</td>
<td>111000</td>
<td>X2</td>
<td>1</td>
<td>xvcvshph</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>VSX Vector Convert with round Single-Precision to Half-Precision format</td>
</tr>
<tr>
<td>111100</td>
<td>111000</td>
<td>X2</td>
<td>1</td>
<td>xxbrr</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>VSX Vector Byte-Reverse Quadword</td>
</tr>
<tr>
<td>111100</td>
<td>111000</td>
<td>X4</td>
<td>1</td>
<td>xxsx</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>VSX Vector Select</td>
</tr>
<tr>
<td>111100</td>
<td>111111</td>
<td>DS</td>
<td>1</td>
<td>sltddp</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>Store Floating Double Pair</td>
</tr>
<tr>
<td>111100</td>
<td>111111</td>
<td>DS</td>
<td>1</td>
<td>lxv</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>Load VSX Vector</td>
</tr>
<tr>
<td>111100</td>
<td>111111</td>
<td>DS</td>
<td>1</td>
<td>stoxd</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>Store VSX Scalar Doubleword</td>
</tr>
<tr>
<td>111100</td>
<td>111111</td>
<td>DS</td>
<td>1</td>
<td>stoxsp</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>Store VSX Scalar Single-Precision</td>
</tr>
<tr>
<td>111100</td>
<td>111111</td>
<td>DS</td>
<td>1</td>
<td>stov</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>Store VSX Vector</td>
</tr>
<tr>
<td>111100</td>
<td>111111</td>
<td>DS</td>
<td>1</td>
<td>std</td>
<td>PPC</td>
<td></td>
<td></td>
<td>Store Doubleword</td>
</tr>
<tr>
<td>111100</td>
<td>111111</td>
<td>DS</td>
<td>1</td>
<td>sldu</td>
<td>PPC</td>
<td></td>
<td></td>
<td>Store Doubleword with Update</td>
</tr>
<tr>
<td>111100</td>
<td>111111</td>
<td>DS</td>
<td>1</td>
<td>slq</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>Store Quadword</td>
</tr>
<tr>
<td>111100</td>
<td>111111</td>
<td>X</td>
<td>1</td>
<td>fcmpu</td>
<td>P1</td>
<td></td>
<td></td>
<td>Floating Compare Unordered</td>
</tr>
<tr>
<td>111100</td>
<td>111111</td>
<td>X</td>
<td>1</td>
<td>fcmpo</td>
<td>P1</td>
<td></td>
<td></td>
<td>Floating Compare Ordered</td>
</tr>
<tr>
<td>111100</td>
<td>111111</td>
<td>X</td>
<td>1</td>
<td>mcfrs</td>
<td>P1</td>
<td></td>
<td></td>
<td>Move To CR from FPSCR</td>
</tr>
<tr>
<td>111100</td>
<td>111111</td>
<td>X</td>
<td>1</td>
<td>ftldv</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>Floating Test for software Divide</td>
</tr>
<tr>
<td>111100</td>
<td>111111</td>
<td>X</td>
<td>1</td>
<td>fltsq</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>Floating Test for software Square Root</td>
</tr>
<tr>
<td>111100</td>
<td>111111</td>
<td>X</td>
<td>1</td>
<td>ddadq</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>DFP Add Quad</td>
</tr>
<tr>
<td>111100</td>
<td>111111</td>
<td>X</td>
<td>1</td>
<td>dmulq</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>DFP Multiply Quad</td>
</tr>
<tr>
<td>111100</td>
<td>111111</td>
<td>X</td>
<td>1</td>
<td>disclq</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>DFP Shift Significand Left Immediate Quad</td>
</tr>
<tr>
<td>111100</td>
<td>111111</td>
<td>X</td>
<td>1</td>
<td>discoq</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>DFP Shift Significand Right Immediate Quad</td>
</tr>
<tr>
<td>111100</td>
<td>111111</td>
<td>X</td>
<td>1</td>
<td>dcmpq</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>DFP Compare Ordered</td>
</tr>
<tr>
<td>111100</td>
<td>111111</td>
<td>X</td>
<td>1</td>
<td>distexq</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>DFP Test Exponent Quad</td>
</tr>
<tr>
<td>111100</td>
<td>111111</td>
<td>X</td>
<td>1</td>
<td>dstdsq</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>DFP Test Data Class Quad</td>
</tr>
<tr>
<td>111100</td>
<td>111111</td>
<td>X</td>
<td>1</td>
<td>dsdstsq</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>DFP Test Data Group Quad</td>
</tr>
<tr>
<td>111100</td>
<td>111111</td>
<td>X</td>
<td>1</td>
<td>dlcmpgpq</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>DFP Convert To DFP Extended</td>
</tr>
<tr>
<td>111100</td>
<td>111111</td>
<td>X</td>
<td>1</td>
<td>dcdfixd</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>DFP Convert To Fixed Quad</td>
</tr>
<tr>
<td>111100</td>
<td>111111</td>
<td>X</td>
<td>1</td>
<td>ddeoptd</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>DFP Decode DPD To BCD Quad</td>
</tr>
<tr>
<td>111100</td>
<td>111111</td>
<td>X</td>
<td>1</td>
<td>dxsexq</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>DFP Extract Exponent Quad</td>
</tr>
<tr>
<td>111100</td>
<td>111111</td>
<td>X</td>
<td>1</td>
<td>dsqubq</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>DFP Subtract Quad</td>
</tr>
<tr>
<td>111100</td>
<td>111111</td>
<td>X</td>
<td>1</td>
<td>ddilvq</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>DFP Divide Quad</td>
</tr>
<tr>
<td>111100</td>
<td>111111</td>
<td>X</td>
<td>1</td>
<td>dcmpvq</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>DFP Compare Unordered Quad</td>
</tr>
<tr>
<td>111100</td>
<td>111111</td>
<td>X</td>
<td>1</td>
<td>dstsdq</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>DFP Test Significance Quad</td>
</tr>
<tr>
<td>111100</td>
<td>111111</td>
<td>X</td>
<td>1</td>
<td>drdipq</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>DFP Round To DFP Long</td>
</tr>
<tr>
<td>111100</td>
<td>111111</td>
<td>X</td>
<td>1</td>
<td>dfrxxq</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>DFP Convert From Fixed Quad</td>
</tr>
<tr>
<td>111100</td>
<td>111111</td>
<td>X</td>
<td>1</td>
<td>dencovq</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>DFP Encode BCD To DPD Quad</td>
</tr>
<tr>
<td>111100</td>
<td>111111</td>
<td>X</td>
<td>1</td>
<td>dinitexq</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>DFP Insert Exponent Quad</td>
</tr>
<tr>
<td>111100</td>
<td>111111</td>
<td>ZZ3</td>
<td>1</td>
<td>dqsdq</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>DFP Quantize Quad</td>
</tr>
<tr>
<td>111100</td>
<td>111111</td>
<td>ZZ3</td>
<td>1</td>
<td>dmdsdq</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>DFP Reround Quad</td>
</tr>
</tbody>
</table>

Figure 88. Power ISA AS Instruction Set Sorted by Opcode (Sheet 16 of 18)
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>Book</th>
<th>Page</th>
<th>Mnemonic</th>
<th>Version</th>
<th>Privilege</th>
<th>Mode</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Z23</td>
<td>203</td>
<td>dqausq</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>DFP Quantize Immediate Quad</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Z23</td>
<td>209</td>
<td>dnsintsq</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>DFP Round To FP Integer With Inexact Quad</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Z23</td>
<td>211</td>
<td>drintsq</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>DFP Round To FP Integer Without Inexact Quad</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
<td>202</td>
<td>dstislq</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>DFP Test Significance Immediate Quad</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
<td>520</td>
<td>xsaddwq</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>VSX Scalar Add Quad-Precision [with round to Odd]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
<td>564</td>
<td>xsdivwq</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>VSX Scalar Divide Quad-Precision [with round to Odd]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
<td>532</td>
<td>xscmpwq</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>VSX Scalar Compare Unordered Quad-Precision</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
<td>654</td>
<td>xstdioq</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>VSX Scalar Test Data Class Quad-Precision</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
<td>512</td>
<td>xsasq</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>VSX Scalar Absolute Quad-Precision</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
<td>656</td>
<td>xsexpq</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>VSX Scalar Extract Exponent Quad-Precision</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
<td>606</td>
<td>xsnsq</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>VSX Scalar Negate Quad-Precision</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
<td>657</td>
<td>xsseiq</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>VSX Scalar Negate Quad-Precision</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
<td>642</td>
<td>xsstpq</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>VSX Scalar Square Root Quad-Precision [with round to Odd]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
<td>554</td>
<td>xsconvwq</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>VSX Scalar Convert with round to zero Quad-Precision to Unsigned Word format</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
<td>560</td>
<td>xsrsvdwp</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>VSX Scalar Convert Signed Word to Quad-Precision format</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
<td>550</td>
<td>xsconvwq</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>VSX Scalar Convert Signed Word format</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
<td>556</td>
<td>xscvsvq</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>VSX Scalar Convert Signed Doubleword to Quad-Precision format</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
<td>552</td>
<td>xscvvpq</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>VSX Scalar Convert with round to zero Quad-Precision to Signed Doubleword format</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
<td>547</td>
<td>xscvvpq</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>VSX Scalar Convert with round to zero Quad-Precision to Signed Doubleword format</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
<td>535</td>
<td>xscvvpq</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>VSX Scalar Convert Double-Precision to Quad-Precision format</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
<td>548</td>
<td>xscvsvq</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>VSX Scalar Convert with round to zero Quad-Precision to Signed Doubleword format</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
<td>569</td>
<td>xscvsvq</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>VSX Scalar Convert Signed Word format</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
<td>634</td>
<td>xsrqpvq</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>VSX Scalar Convert Signed Word with round to Odd format to Word format</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
<td>615</td>
<td>xscvsvq</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>VSX Scalar Convert Signed Word with round to Odd format</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
<td>175</td>
<td>mtfsbq</td>
<td>P1</td>
<td></td>
<td></td>
<td>Move To FPSCR Bit 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
<td>175</td>
<td>mtfsbq</td>
<td>P1</td>
<td></td>
<td></td>
<td>Move To FPSCR Bit 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
<td>172</td>
<td>mtfsbq</td>
<td>P1</td>
<td></td>
<td></td>
<td>Move To FPSCR Field Immediate</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
<td>151</td>
<td>fmrgw</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>Floating Merge Odd Word</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
<td>151</td>
<td>fmrgw</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>Floating Merge Even Word</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
<td>170</td>
<td>mtfsiq</td>
<td>P1</td>
<td></td>
<td></td>
<td>Move From FPSCR</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
<td>170</td>
<td>mtfsiq</td>
<td>P1</td>
<td></td>
<td></td>
<td>Move From FPSCR</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
<td>170</td>
<td>mtfsiq</td>
<td>P1</td>
<td></td>
<td></td>
<td>Move From FPSCR &amp; Clear Enables</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
<td>170</td>
<td>mtfsaq</td>
<td>v3.0B</td>
<td></td>
<td></td>
<td>Move From FPSCR Control &amp; set DRN</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
<td>170</td>
<td>mtfsaq</td>
<td>v3.0B</td>
<td></td>
<td></td>
<td>Move From FPSCR Control &amp; set DRN Immediate</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
<td>170</td>
<td>mtfsaq</td>
<td>v3.0B</td>
<td></td>
<td></td>
<td>Move From FPSCR Control &amp; set RN</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
<td>170</td>
<td>mtfsaq</td>
<td>v3.0B</td>
<td></td>
<td></td>
<td>Move From FPSCR Control &amp; set RN Immediate</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
<td>170</td>
<td>mtsisi</td>
<td>v3.0B</td>
<td></td>
<td></td>
<td>Move From FPSCR Lightweight</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
<td>172</td>
<td>mtfsbq</td>
<td>P1</td>
<td></td>
<td></td>
<td>Move To FPSCR Fields</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
<td>150</td>
<td>kpsiq</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>Floating Copy Sign</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
<td>150</td>
<td>inetiq</td>
<td>P1</td>
<td></td>
<td></td>
<td>Floating Negate</td>
</tr>
</tbody>
</table>

Figure 88. Power ISA AS Instruction Set Sorted by Opcode (Sheet 17 of 18)
### Figure 88. Power ISA AS Instruction Set Sorted by Opcode (Sheet 18 of 18)

<table>
<thead>
<tr>
<th>Instruction&lt;sup&gt;1&lt;/sup&gt;</th>
<th>Format</th>
<th>Book</th>
<th>Page</th>
<th>Mnemonic</th>
<th>Version&lt;sup&gt;2&lt;/sup&gt;</th>
<th>Privilege&lt;sup&gt;3&lt;/sup&gt;</th>
<th>Mode Dep&lt;sup&gt;4&lt;/sup&gt;</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:5 6:30 13:15 16:20 21:25 26:30</td>
<td>X</td>
<td>150</td>
<td>fnrt[.]</td>
<td>P1</td>
<td>Floating Move Register</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0010 00100.</td>
<td>X</td>
<td>150</td>
<td>fnabs[.]</td>
<td>P1</td>
<td>Floating Negative Absolute Value</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0100 01000.</td>
<td>X</td>
<td>150</td>
<td>fadd[.]</td>
<td>P1</td>
<td>Floating Absolute</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0110 01000.</td>
<td>X</td>
<td>150</td>
<td>fadd[.]</td>
<td>P2</td>
<td>Floating Round To Integer Nearest</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0111 01000.</td>
<td>X</td>
<td>150</td>
<td>fadd[.]</td>
<td>v2.02</td>
<td>Floating Round To Integer Zero</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0111 01000.</td>
<td>X</td>
<td>150</td>
<td>fadd[.]</td>
<td>v2.02</td>
<td>Floating Round To Integer Plus</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0111 01000.</td>
<td>X</td>
<td>150</td>
<td>fadd[.]</td>
<td>v2.02</td>
<td>Floating Round To Integer Minus</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0111 01000.</td>
<td>X</td>
<td>150</td>
<td>fadd[.]</td>
<td>P1</td>
<td>Floating Round To Single-Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>161</td>
<td>fct2w[.]</td>
<td>P2</td>
<td>Floating Convert with round Double-Precision To Signed Word format</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0010 01110.</td>
<td>X</td>
<td>162</td>
<td>fct2w[.]</td>
<td>v2.06</td>
<td>Floating Convert with round Double-Precision To UNSIGNED Word format</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1101 01110.</td>
<td>X</td>
<td>159</td>
<td>fct2d[.]</td>
<td>PPC</td>
<td>Floating Convert with round Double-Precision To Signed Doubleword format</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1101 01110.</td>
<td>X</td>
<td>163</td>
<td>fct2d[.]</td>
<td>PPC</td>
<td>Floating Convert with round Signed Doubleword to Double-Precision format</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1101 01110.</td>
<td>X</td>
<td>160</td>
<td>fct2d[.]</td>
<td>v2.06</td>
<td>Floating Convert with round Double-Precision To UNSIGNED Doubleword format</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1101 01110.</td>
<td>X</td>
<td>160</td>
<td>fct2d[.]</td>
<td>v2.06</td>
<td>Floating Convert with round UNSIGNED Doubleword to Double-Precision format</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>162</td>
<td>fct2d[.]</td>
<td>P2</td>
<td>Floating Convert with round to Zero Double-Precision To Signed Word format</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>162</td>
<td>fct2w[.]</td>
<td>v2.06</td>
<td>Floating Convert with round to Zero Double-Precision To UNSIGNED Word format</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1101 01110.</td>
<td>X</td>
<td>160</td>
<td>fct2d[.]</td>
<td>PPC</td>
<td>Floating Convert with round to Zero Double-Precision To Signed Doubleword format</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>161</td>
<td>fct2d[.]</td>
<td>v2.06</td>
<td>Floating Convert with round to Zero Double-Precision To UNSIGNED Doubleword format</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>153</td>
<td>fdiv[.]</td>
<td>P1</td>
<td>Floating Divide</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1100</td>
<td>A</td>
<td>152</td>
<td>fsub[.]</td>
<td>P1</td>
<td>Floating Subtract</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1101</td>
<td>A</td>
<td>152</td>
<td>fadd[.]</td>
<td>P1</td>
<td>Floating Add</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1100</td>
<td>A</td>
<td>154</td>
<td>fsec[.]</td>
<td>PPC</td>
<td>Floating Square Root</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1101</td>
<td>A</td>
<td>168</td>
<td>fsel[.]</td>
<td>PPC</td>
<td>Floating Select</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1100</td>
<td>A</td>
<td>154</td>
<td>frec[.]</td>
<td>v2.02</td>
<td>Floating Reciprocal Estimate</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1101</td>
<td>A</td>
<td>153</td>
<td>fmul[.]</td>
<td>P1</td>
<td>Floating Multiply</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1101</td>
<td>A</td>
<td>155</td>
<td>fmg[.]</td>
<td>PPC</td>
<td>Floating Reciprocal Square Root Estimate</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1100</td>
<td>A</td>
<td>158</td>
<td>fmsub[.]</td>
<td>P1</td>
<td>Floating Multiply-Subtract</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1100</td>
<td>A</td>
<td>157</td>
<td>fma[.]</td>
<td>P1</td>
<td>Floating Multiply-Add</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1100</td>
<td>A</td>
<td>158</td>
<td>fmsub[.]</td>
<td>P1</td>
<td>Floating Negative Multiply-Subtract</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1100</td>
<td>A</td>
<td>158</td>
<td>fma[.]</td>
<td>P1</td>
<td>Floating Negative Multiply-Add</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

1. Key to instruction column.

   - 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 having a value 0.
   - Instruction bit having a value 1.

2. Key to Version column.

   - P1 Instruction introduced in the POWER Architecture.
   - P2 Instruction introduced in the POWER2 Architecture.
   - PPC Instruction introduced in the PowerPC Architecture prior to v2.00.
   - v2.00 Instruction introduced in the PowerPC Architecture Version 2.00.
   - v2.01 Instruction introduced in the PowerPC Architecture Version 2.01.
   - v2.02 Instruction introduced in the PowerPC Architecture Version 2.02.
   - v2.03 Instruction introduced in the Power ISA Architecture Version 2.03.
   - v2.04 Instruction introduced in the Power ISA Architecture Version 2.04.
   - v2.05 Instruction introduced in the Power ISA Architecture Version 2.05.
   - v2.06 Instruction introduced in the Power ISA Architecture Version 2.06.
   - v2.07 Instruction introduced in the Power ISA Architecture Version 2.07.
   - v3.0 Instruction introduced in the Power ISA Architecture Version 3.0.
   - v3.0B Instruction introduced in the Power ISA Architecture Version 3.0B.
3. Key to Privilege column.

- **P** Denotes an instruction that is treated as privileged.
- **O** Denotes an instruction that is treated as privileged or nonprivileged (or hypervisor, for mtspr), depending on the SPR or PMR number.
- **PI** Denotes an instruction that is illegal in privileged state.
- **H** Denotes an instruction that can be executed only in hypervisor state
- **U** Denotes an instruction that can be executed only in ultravisor state

4. Key to Mode Dependency column.
   Except as described below and in Section 1.11.3, “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.
- **32** The instruction can be executed only in 32-bit mode.
- **64** The instruction can be executed only in 64-bit mode.
Appendix E. Power ISA Instruction Set Sorted by Version

This appendix lists all the instructions in the Power ISA, sorted in reverse order by ISA version.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>Book</th>
<th>Page</th>
<th>Mnemonic</th>
<th>Version</th>
<th>Privilege</th>
<th>Mode</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>72</td>
<td>X</td>
<td>1</td>
<td>72</td>
<td>addex</td>
<td>v3.0B</td>
<td></td>
<td></td>
<td>Add Extended using alternate carry</td>
</tr>
<tr>
<td>170</td>
<td>X</td>
<td>1</td>
<td>170</td>
<td>mffscdm</td>
<td>v3.0B</td>
<td></td>
<td></td>
<td>Move From FPSCR Control &amp; set DRN</td>
</tr>
<tr>
<td>170</td>
<td>X</td>
<td>1</td>
<td>170</td>
<td>mffscdmi</td>
<td>v3.0B</td>
<td></td>
<td></td>
<td>Move From FPSCR Control &amp; set DRN Immediate</td>
</tr>
<tr>
<td>170</td>
<td>X</td>
<td>1</td>
<td>170</td>
<td>mffscdir</td>
<td>v3.0B</td>
<td></td>
<td></td>
<td>Move From FPSCR Control &amp; Clear Enables</td>
</tr>
<tr>
<td>170</td>
<td>X</td>
<td>1</td>
<td>170</td>
<td>mffscrn</td>
<td>v3.0B</td>
<td></td>
<td></td>
<td>Move From FPSCR Control &amp; set RN</td>
</tr>
<tr>
<td>170</td>
<td>X</td>
<td>1</td>
<td>170</td>
<td>mffscri</td>
<td>v3.0B</td>
<td></td>
<td></td>
<td>Move From FPSCR Control &amp; set RN Immediate</td>
</tr>
<tr>
<td>170</td>
<td>X</td>
<td>1</td>
<td>170</td>
<td>mffscl</td>
<td>v3.0B</td>
<td></td>
<td></td>
<td>Move From FPSCR Lightweight</td>
</tr>
<tr>
<td>159</td>
<td>X</td>
<td>1</td>
<td>159</td>
<td>slbiag</td>
<td>v3.0B</td>
<td></td>
<td></td>
<td>Vector Multiply-Sum Unsigned Doubleword Modulo</td>
</tr>
<tr>
<td>289</td>
<td>VA</td>
<td>1</td>
<td>289</td>
<td>vmsumudm</td>
<td>v3.0B</td>
<td></td>
<td></td>
<td>Add PC Immediate Shifted</td>
</tr>
<tr>
<td>350</td>
<td>VX</td>
<td>1</td>
<td>350</td>
<td>bcdcfn.</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>Decimal Convert From National &amp; record</td>
</tr>
<tr>
<td>354</td>
<td>VX</td>
<td>1</td>
<td>354</td>
<td>bcdtsq.</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>Decimal Convert From Signed Quadword &amp; record</td>
</tr>
<tr>
<td>351</td>
<td>VX</td>
<td>1</td>
<td>351</td>
<td>bcdtz.</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>Decimal Convert From Zoned &amp; record</td>
</tr>
<tr>
<td>356</td>
<td>VX</td>
<td>1</td>
<td>356</td>
<td>bcdtsign.</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>Decimal Copy/Sign &amp; record</td>
</tr>
<tr>
<td>352</td>
<td>VX</td>
<td>1</td>
<td>352</td>
<td>bcdln.</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>Decimal Convert To National &amp; record</td>
</tr>
<tr>
<td>354</td>
<td>VX</td>
<td>1</td>
<td>354</td>
<td>bcdtsq.</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>Decimal Convert To Signed Quadword &amp; record</td>
</tr>
<tr>
<td>353</td>
<td>VX</td>
<td>1</td>
<td>353</td>
<td>bcdtz.</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>Decimal Convert To Zoned &amp; record</td>
</tr>
<tr>
<td>357</td>
<td>VX</td>
<td>1</td>
<td>357</td>
<td>bcds.</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>Decimal Shift &amp; record</td>
</tr>
<tr>
<td>356</td>
<td>VX</td>
<td>1</td>
<td>356</td>
<td>bcdsetgn.</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>Decimal Set Sign &amp; record</td>
</tr>
<tr>
<td>359</td>
<td>VX</td>
<td>1</td>
<td>359</td>
<td>bcdar.</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>Decimal Shift &amp; Round &amp; record</td>
</tr>
<tr>
<td>360</td>
<td>VX</td>
<td>1</td>
<td>360</td>
<td>bcdtrunc.</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>Decimal Truncate &amp; record</td>
</tr>
<tr>
<td>358</td>
<td>VX</td>
<td>1</td>
<td>358</td>
<td>bcdus.</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>Decimal Unsigned Shift &amp; record</td>
</tr>
<tr>
<td>361</td>
<td>VX</td>
<td>1</td>
<td>361</td>
<td>bcdultrunc.</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>Decimal Unsigned Truncate &amp; record</td>
</tr>
<tr>
<td>99</td>
<td>X</td>
<td>1</td>
<td>99</td>
<td>cnttze[n]</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>Count Trailing Zeros Doubleword</td>
</tr>
<tr>
<td>96</td>
<td>X</td>
<td>1</td>
<td>96</td>
<td>cnttzw[n]</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>Count Trailing Zeros Word</td>
</tr>
<tr>
<td>855</td>
<td>X</td>
<td>1</td>
<td>855</td>
<td>copy</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>Copy</td>
</tr>
<tr>
<td>856</td>
<td>X</td>
<td>1</td>
<td>856</td>
<td>cp_abort</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>CP_Abort</td>
</tr>
<tr>
<td>78</td>
<td>X</td>
<td>1</td>
<td>78</td>
<td>dam</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>Deliver A Random Number</td>
</tr>
<tr>
<td>202</td>
<td>X</td>
<td>1</td>
<td>202</td>
<td>dstsfl</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>DFP Test Significance Immediate</td>
</tr>
<tr>
<td>202</td>
<td>X</td>
<td>1</td>
<td>202</td>
<td>dstsfr</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>DFP Test Significance Immediate Quadr</td>
</tr>
<tr>
<td>110</td>
<td>XS</td>
<td>1</td>
<td>110</td>
<td>extsw[.]</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>Extend Sign Word &amp; Shift Left Immediate</td>
</tr>
<tr>
<td>860</td>
<td>X</td>
<td>1</td>
<td>860</td>
<td>ldtd</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>Load Doubleword Atomic</td>
</tr>
<tr>
<td>860</td>
<td>X</td>
<td>1</td>
<td>860</td>
<td>ldwa</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>Load Word Atomic</td>
</tr>
<tr>
<td>480</td>
<td>DS</td>
<td>1</td>
<td>480</td>
<td>lvx</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>Load VSX Scalar Doubleword</td>
</tr>
<tr>
<td>482</td>
<td>DS</td>
<td>1</td>
<td>482</td>
<td>lvshx</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>Load VSX Scalar as Integer Byte &amp; Zero Indexed</td>
</tr>
<tr>
<td>482</td>
<td>DS</td>
<td>1</td>
<td>482</td>
<td>lvshzv</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>Load VSX Scalar as Integer Halfword &amp; Zero Indexed</td>
</tr>
<tr>
<td>485</td>
<td>DS</td>
<td>1</td>
<td>485</td>
<td>lvssap</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>Load VSX Scalar Single</td>
</tr>
<tr>
<td>492</td>
<td>DQ</td>
<td>1</td>
<td>492</td>
<td>lvxv</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>Load VSX Vector</td>
</tr>
<tr>
<td>487</td>
<td>DQ</td>
<td>1</td>
<td>487</td>
<td>lvxv16x</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>Load VSX Vector Byte*16 Indexed</td>
</tr>
</tbody>
</table>

Figure 89. Power ISA AS Instruction Set Sorted by Version (Sheet 1 of 18)
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>Book</th>
<th>Page</th>
<th>Mnemonic</th>
<th>Version</th>
<th>Privilege</th>
<th>Mode Dep</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>
</tr>
</tbody>
</table>

Figure 89. Power ISA AS Instruction Set Sorted by Version (Sheet 2 of 18)

1200 Power ISA™ Appendices
<table>
<thead>
<tr>
<th>Instruction1</th>
<th>Format</th>
<th>Book</th>
<th>Page</th>
<th>Mnemonic</th>
<th>Version2</th>
<th>Privilege3</th>
<th>Mode Dep4</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0110</td>
<td>VX</td>
<td>I</td>
<td>294</td>
<td>vecxtsb2d</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0110</td>
<td>VX</td>
<td>I</td>
<td>294</td>
<td>vecxtsb2w</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0110</td>
<td>VX</td>
<td>I</td>
<td>294</td>
<td>vecxtsh2d</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0110</td>
<td>VX</td>
<td>I</td>
<td>294</td>
<td>vecxtsh2w</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0110</td>
<td>VX</td>
<td>I</td>
<td>294</td>
<td>vecxtsw2d</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0110</td>
<td>VX</td>
<td>I</td>
<td>294</td>
<td>vecxtwhl</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0110</td>
<td>VX</td>
<td>I</td>
<td>294</td>
<td>vecxublx</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0110</td>
<td>VX</td>
<td>I</td>
<td>294</td>
<td>vecxubnx</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0110</td>
<td>VX</td>
<td>I</td>
<td>294</td>
<td>vecxulqu</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0110</td>
<td>VX</td>
<td>I</td>
<td>294</td>
<td>vecxumqu</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0110</td>
<td>VX</td>
<td>I</td>
<td>288</td>
<td>vinsrdb</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0110</td>
<td>VX</td>
<td>I</td>
<td>288</td>
<td>vinsrtd</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0110</td>
<td>VX</td>
<td>I</td>
<td>288</td>
<td>vinsrth</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0110</td>
<td>VX</td>
<td>I</td>
<td>288</td>
<td>vinsrtw</td>
<td>v3.0</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>0110</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>0110</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>0110</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>0110</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>0110</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>0110</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>
</tbody>
</table>

Figure 89. Power ISA AS Instruction Set Sorted by Version (Sheet 3 of 18)
<table>
<thead>
<tr>
<th>Instruction(^1)</th>
<th>Format</th>
<th>Page</th>
<th>Mnemonic (2)</th>
<th>Version (3)</th>
<th>Privilege (4)</th>
<th>Mode Dep (5)</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<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>
</tr>
<tr>
<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>
</tr>
<tr>
<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>
</tr>
<tr>
<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>
</tr>
<tr>
<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>
</tr>
</tbody>
</table>

Figure 89. Power ISA AS Instruction Set Sorted by Version (Sheet 4 of 18)

Power ISA™ Appendices
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>Book</th>
<th>Page</th>
<th>Mnemonic</th>
<th>Privilege</th>
<th>Mode</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<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>
</tr>
</tbody>
</table>

**Figure 89. Power ISA AS Instruction Set Sorted by Version (Sheet 5 of 18)**

**Appendix E. Power ISA Instruction Set Sorted by Version**

1203
<table>
<thead>
<tr>
<th>Instruction¹</th>
<th>Format</th>
<th>Book</th>
<th>Page</th>
<th>Mnemonic</th>
<th>Version²</th>
<th>Privilege³</th>
<th>Mode Dep⁴</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>00100</td>
<td>VX</td>
<td>L</td>
<td>283</td>
<td>vmulouw</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>Vector Multiply Odd Unsigned Word</td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>L</td>
<td>284</td>
<td>vmuluw</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>Vector Multiply Unsigned Word Modulo</td>
</tr>
<tr>
<td>01010</td>
<td>VX</td>
<td>L</td>
<td>312</td>
<td>vnand</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>Vector NAND</td>
</tr>
<tr>
<td>01011</td>
<td>VX</td>
<td>L</td>
<td>334</td>
<td>vnipicer</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>Vector AES Inverse Cipher</td>
</tr>
<tr>
<td>01011</td>
<td>VX</td>
<td>L</td>
<td>334</td>
<td>vnipicerlast</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>Vector AES Inverse Cipher Last</td>
</tr>
<tr>
<td>01011</td>
<td>VX</td>
<td>L</td>
<td>313</td>
<td>vorc</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>Vector OR with Complement</td>
</tr>
<tr>
<td>01011</td>
<td>VX</td>
<td>L</td>
<td>338</td>
<td>vperrmceor</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>Vector Permute &amp; Exclusive-OR</td>
</tr>
<tr>
<td>01011</td>
<td>VX</td>
<td>L</td>
<td>248</td>
<td>vpkddas</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>Vector Pack Signed Doubleword Signed Saturate</td>
</tr>
<tr>
<td>01011</td>
<td>VX</td>
<td>L</td>
<td>248</td>
<td>vpkddus</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>Vector Pack Signed Doubleword Unsigned Saturate</td>
</tr>
<tr>
<td>01011</td>
<td>VX</td>
<td>L</td>
<td>261</td>
<td>vpktdum</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>Vector Pack Unsigned Doubleword Unsigned Modulo</td>
</tr>
<tr>
<td>01011</td>
<td>VX</td>
<td>L</td>
<td>261</td>
<td>vpktdus</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>Vector Pack Unsigned Doubleword Unsigned Saturate</td>
</tr>
<tr>
<td>01011</td>
<td>VX</td>
<td>L</td>
<td>336</td>
<td>vpsumb</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>Vector Polynomial Multiply-Sum Byte</td>
</tr>
<tr>
<td>01011</td>
<td>VX</td>
<td>L</td>
<td>336</td>
<td>vpsumad</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>Vector Polynomial Multiply-Sum Doubleword</td>
</tr>
<tr>
<td>01011</td>
<td>VX</td>
<td>L</td>
<td>336</td>
<td>vpsumah</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>Vector Polynomial Multiply-Sum Halfword</td>
</tr>
<tr>
<td>01011</td>
<td>VX</td>
<td>L</td>
<td>336</td>
<td>vpsumaw</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>Vector Polynomial Multiply-Sum Word</td>
</tr>
<tr>
<td>00001</td>
<td>VX</td>
<td>L</td>
<td>315</td>
<td>vrd</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>Vector Rotate Left Doubleword</td>
</tr>
<tr>
<td>01001</td>
<td>VX</td>
<td>L</td>
<td>334</td>
<td>vsbox</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>Vector AES S-Box</td>
</tr>
<tr>
<td>01101</td>
<td>VX</td>
<td>L</td>
<td>335</td>
<td>vshasigmad</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>Vector SHA-512 Sigma Doubleword</td>
</tr>
<tr>
<td>01101</td>
<td>VX</td>
<td>L</td>
<td>335</td>
<td>vshasigmaw</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>Vector SHA-256 Sigma Word</td>
</tr>
<tr>
<td>01101</td>
<td>VX</td>
<td>L</td>
<td>341</td>
<td>vvalid</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>Vector Shift Left Doubleword</td>
</tr>
<tr>
<td>01101</td>
<td>VX</td>
<td>L</td>
<td>318</td>
<td>vsrad</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>Vector Shift Right Algebraic Doubleword</td>
</tr>
<tr>
<td>01101</td>
<td>VX</td>
<td>L</td>
<td>317</td>
<td>vsrdd</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>Vector Shift Right Doubleword</td>
</tr>
<tr>
<td>01101</td>
<td>VX</td>
<td>L</td>
<td>297</td>
<td>vsubucq</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>Vector Subtract &amp; write Carry Signed Quadword</td>
</tr>
<tr>
<td>01101</td>
<td>VX</td>
<td>L</td>
<td>297</td>
<td>vsubecq</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>Vector Subtract Extended &amp; write Carry Unsigned Quadword</td>
</tr>
<tr>
<td>01001</td>
<td>VX</td>
<td>L</td>
<td>277</td>
<td>vsubdum</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>Vector Subtract Unsigned Doubleword Modulo</td>
</tr>
<tr>
<td>10100</td>
<td>VX</td>
<td>L</td>
<td>279</td>
<td>vsubqum</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>Vector Subtract Unsigned Quadword Modulo</td>
</tr>
<tr>
<td>10101</td>
<td>VX</td>
<td>L</td>
<td>254</td>
<td>vupkhhw</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>Vector Unpack High Signed Word</td>
</tr>
<tr>
<td>10101</td>
<td>VX</td>
<td>L</td>
<td>254</td>
<td>vupklsw</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>Vector Unpack Low Signed Word</td>
</tr>
<tr>
<td>00000</td>
<td>XX</td>
<td></td>
<td>518</td>
<td>xsaddsp</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>VSX Scalar Add Single-Precision</td>
</tr>
<tr>
<td>10000</td>
<td>XX</td>
<td></td>
<td>537</td>
<td>xscvdspsn</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>VSX Scalar Convert Double-Precision to Single-Precision Non-signalling format</td>
</tr>
<tr>
<td>10100</td>
<td>XX</td>
<td></td>
<td>558</td>
<td>xscvdsdpn</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>VSX Scalar Convert Single-Precision to Double-Precision Non-signalling format</td>
</tr>
<tr>
<td>10101</td>
<td>XX</td>
<td></td>
<td>559</td>
<td>xscvdxsdp</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>VSX Scalar Convert withRound Signed Doubleword to Single-Precision format</td>
</tr>
<tr>
<td>10100</td>
<td>XX</td>
<td></td>
<td>561</td>
<td>xscvuxsdp</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>VSX Scalar Convert withRound Unsigned Doubleword to Single-Precision format</td>
</tr>
<tr>
<td>00011</td>
<td>XX</td>
<td></td>
<td>566</td>
<td>xsvdvsp</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>VSX Scalar Divide Single-Precision</td>
</tr>
<tr>
<td>00001</td>
<td>XX</td>
<td></td>
<td>573</td>
<td>xsmaddasp</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>VSX Scalar Multiply-Add Type-A Single-Precision</td>
</tr>
<tr>
<td>00011</td>
<td>XX</td>
<td></td>
<td>573</td>
<td>xsmaddmpsp</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>VSX Scalar Multiply-Add Type-M Single-Precision</td>
</tr>
<tr>
<td>00010</td>
<td>XX</td>
<td></td>
<td>594</td>
<td>xsmsubasp</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>VSX Scalar Multiply-Subtract Type-A Single-Precision</td>
</tr>
<tr>
<td>00010</td>
<td>XX</td>
<td></td>
<td>594</td>
<td>xsmsubmpsp</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>VSX Scalar Multiply-Subtract Type-M Single-Precision</td>
</tr>
<tr>
<td>00000</td>
<td>XX</td>
<td></td>
<td>604</td>
<td>xsmulpsp</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>VSX Scalar Multiply Single-Precision</td>
</tr>
<tr>
<td>01000</td>
<td>XX</td>
<td></td>
<td>613</td>
<td>xsmnaddasp</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>VSX Scalar Negative Multiply-Add Type-A Single-Precision</td>
</tr>
<tr>
<td>01000</td>
<td>XX</td>
<td></td>
<td>613</td>
<td>xsmnaddmpsp</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>VSX Scalar Negative Multiply-Add Type-M Single-Precision</td>
</tr>
<tr>
<td>01000</td>
<td>XX</td>
<td></td>
<td>622</td>
<td>xsmnsubasp</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>VSX Scalar Negative Multiply-Subtract Type-A Single-Precision</td>
</tr>
<tr>
<td>01000</td>
<td>XX</td>
<td></td>
<td>622</td>
<td>xsmnsubmpsp</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>VSX Scalar Negative Multiply-Subtract Type-M Single-Precision</td>
</tr>
<tr>
<td>00000</td>
<td>XX</td>
<td></td>
<td>633</td>
<td>xsresp</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>VSX Scalar Reciprocal Estimate Single-Precision</td>
</tr>
<tr>
<td>00011</td>
<td>XX</td>
<td></td>
<td>638</td>
<td>xsresp</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>VSX Scalar Round Double-Precision to Single-Precision</td>
</tr>
<tr>
<td>00010</td>
<td>XX</td>
<td></td>
<td>640</td>
<td>xsrsqrspl</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>VSX Scalar Round Square Root Estimate Single-Precision</td>
</tr>
<tr>
<td>00000</td>
<td>XX</td>
<td></td>
<td>644</td>
<td>xsrsqrspl</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>VSX Scalar Square Root Single-Precision</td>
</tr>
<tr>
<td>00010</td>
<td>XX</td>
<td></td>
<td>649</td>
<td>xsrsqrspl</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>VSX Scalar Subtract Single-Precision</td>
</tr>
</tbody>
</table>

Figure 89. Power ISA AS Instruction Set Sorted by Version (Sheet 6 of 18)
## Appendix E. Power ISA Instruction Set Sorted by Version

### Table: Power ISA AS Instruction Set Sorted by Version (Sheet 7 of 18)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>Book</th>
<th>Page</th>
<th>Mnemonic</th>
<th>Version</th>
<th>Privilege</th>
<th>Mode Dep</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>311100</td>
<td></td>
<td></td>
<td>768</td>
<td>xorleqv</td>
<td>v2.07</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>311100</td>
<td></td>
<td></td>
<td>768</td>
<td>xorinand</td>
<td>v2.07</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>311100</td>
<td></td>
<td></td>
<td>768</td>
<td>xoror</td>
<td>v2.07</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>311100</td>
<td></td>
<td></td>
<td>768</td>
<td>xoror</td>
<td>v2.07</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>311100</td>
<td></td>
<td></td>
<td>768</td>
<td>xoror</td>
<td>v2.07</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>311100</td>
<td></td>
<td></td>
<td>768</td>
<td>xoror</td>
<td>v2.07</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>311100</td>
<td></td>
<td></td>
<td>768</td>
<td>xoror</td>
<td>v2.07</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>311100</td>
<td></td>
<td></td>
<td>768</td>
<td>xoror</td>
<td>v2.07</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>311100</td>
<td></td>
<td></td>
<td>768</td>
<td>xoror</td>
<td>v2.07</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>311100</td>
<td></td>
<td></td>
<td>768</td>
<td>xoror</td>
<td>v2.07</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>311100</td>
<td></td>
<td></td>
<td>768</td>
<td>xoror</td>
<td>v2.07</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>311100</td>
<td></td>
<td></td>
<td>768</td>
<td>xoror</td>
<td>v2.07</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>311100</td>
<td></td>
<td></td>
<td>768</td>
<td>xoror</td>
<td>v2.07</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>311100</td>
<td></td>
<td></td>
<td>768</td>
<td>xoror</td>
<td>v2.07</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>311100</td>
<td></td>
<td></td>
<td>768</td>
<td>xoror</td>
<td>v2.07</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>311100</td>
<td></td>
<td></td>
<td>768</td>
<td>xoror</td>
<td>v2.07</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>311100</td>
<td></td>
<td></td>
<td>768</td>
<td>xoror</td>
<td>v2.07</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>311100</td>
<td></td>
<td></td>
<td>768</td>
<td>xoror</td>
<td>v2.07</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>311100</td>
<td></td>
<td></td>
<td>768</td>
<td>xoror</td>
<td>v2.07</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>311100</td>
<td></td>
<td></td>
<td>768</td>
<td>xoror</td>
<td>v2.07</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>311100</td>
<td></td>
<td></td>
<td>768</td>
<td>xoror</td>
<td>v2.07</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>311100</td>
<td></td>
<td></td>
<td>768</td>
<td>xoror</td>
<td>v2.07</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>311100</td>
<td></td>
<td></td>
<td>768</td>
<td>xoror</td>
<td>v2.07</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>311100</td>
<td></td>
<td></td>
<td>768</td>
<td>xoror</td>
<td>v2.07</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>311100</td>
<td></td>
<td></td>
<td>768</td>
<td>xoror</td>
<td>v2.07</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>311100</td>
<td></td>
<td></td>
<td>768</td>
<td>xoror</td>
<td>v2.07</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>311100</td>
<td></td>
<td></td>
<td>768</td>
<td>xoror</td>
<td>v2.07</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>311100</td>
<td></td>
<td></td>
<td>768</td>
<td>xoror</td>
<td>v2.07</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>311100</td>
<td></td>
<td></td>
<td>768</td>
<td>xoror</td>
<td>v2.07</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>311100</td>
<td></td>
<td></td>
<td>768</td>
<td>xoror</td>
<td>v2.07</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>311100</td>
<td></td>
<td></td>
<td>768</td>
<td>xoror</td>
<td>v2.07</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>311100</td>
<td></td>
<td></td>
<td>768</td>
<td>xoror</td>
<td>v2.07</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>311100</td>
<td></td>
<td></td>
<td>768</td>
<td>xoror</td>
<td>v2.07</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

**Figure 89.** Power ISA AS Instruction Set Sorted by Version (Sheet 7 of 18)
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Book</th>
<th>Page</th>
<th>Mnemonic</th>
<th>Version</th>
<th>Privilege</th>
<th>Mode Dep</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>XX2</td>
<td>1</td>
<td>557</td>
<td>xscvppdp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>VSX Scalar Convert Single-Precision to Double-Precision format</td>
</tr>
<tr>
<td>XX2</td>
<td>1</td>
<td>559</td>
<td>xscvxxdp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>VSX Scalar Convert with round Signed Doubleword to Double-Precision format</td>
</tr>
<tr>
<td>XX2</td>
<td>1</td>
<td>561</td>
<td>xscvxxdpp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>VSX Scalar Convert with round Unsigned Doubleword to Double-Precision format</td>
</tr>
<tr>
<td>XX3</td>
<td>1</td>
<td>562</td>
<td>xsdvdp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>VSX Scalar Divide Double-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>1</td>
<td>570</td>
<td>xsmaddadp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>VSX Scalar Multiply-Add Type-A Double-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>1</td>
<td>571</td>
<td>xsmaddmdp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>VSX Scalar Multiply-Add Type-M Double-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>1</td>
<td>579</td>
<td>xsmaxdp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>VSX Scalar Maximum Double-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>1</td>
<td>581</td>
<td>xsmindp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>VSX Scalar Minimum Double-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>1</td>
<td>591</td>
<td>xsmsubadp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>VSX Scalar Multiply-Subtract Type-A Double-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>1</td>
<td>591</td>
<td>xsmsubmdp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>VSX Scalar Multiply-Subtract Type-M Double-Precision</td>
</tr>
<tr>
<td>XX1</td>
<td>1</td>
<td>600</td>
<td>xsmuldp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>VSX Scalar Multiply Double-Precision</td>
</tr>
<tr>
<td>XX2</td>
<td>1</td>
<td>606</td>
<td>xsnabdp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>VSX Scalar Negative Absolute Double-Precision</td>
</tr>
<tr>
<td>XX2</td>
<td>1</td>
<td>608</td>
<td>xsnimaddp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>VSX Scalar Negative Multiply-Add Type-A Double-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>1</td>
<td>619</td>
<td>xsmxsubdp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>VSX Scalar Negative Multiply-Subtract Type-A Double-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>1</td>
<td>619</td>
<td>xsmxsubmdp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>VSX Scalar Negative Multiply-Subtract Type-M Double-Precision</td>
</tr>
<tr>
<td>XX2</td>
<td>1</td>
<td>628</td>
<td>xsrdpi</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>VSX Scalar Round Double-Precision to Integer</td>
</tr>
<tr>
<td>XX2</td>
<td>1</td>
<td>629</td>
<td>xsrdpic</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>VSX Scalar Round Double-Precision to Integer using current rounding mode</td>
</tr>
<tr>
<td>XX2</td>
<td>1</td>
<td>630</td>
<td>xsrdpm</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>VSX Scalar Round Double-Precision to Integer toward -Infinity</td>
</tr>
<tr>
<td>XX2</td>
<td>1</td>
<td>631</td>
<td>xsrdpp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>VSX Scalar Round Double-Precision to Integer toward +Infinity</td>
</tr>
<tr>
<td>XX2</td>
<td>1</td>
<td>632</td>
<td>xsrdp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>VSX Scalar Round Double-Precision to Integer toward Zero</td>
</tr>
<tr>
<td>XX2</td>
<td>1</td>
<td>639</td>
<td>xsrqdp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>VSX Scalar Reciprocal Estimate Double-Precision</td>
</tr>
<tr>
<td>XX2</td>
<td>1</td>
<td>641</td>
<td>xsrdp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>VSX Scalar Square Root Double-Precision</td>
</tr>
<tr>
<td>XX2</td>
<td>1</td>
<td>645</td>
<td>xsrdp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>VSX Scalar Subtract Double-Precision</td>
</tr>
<tr>
<td>XX2</td>
<td>1</td>
<td>651</td>
<td>xsdvdp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>VSX Scalar Test for software Divide Double-Precision</td>
</tr>
<tr>
<td>XX2</td>
<td>1</td>
<td>652</td>
<td>xstqdp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>VSX Scalar Test for software Square Root Double-Precision</td>
</tr>
<tr>
<td>XX2</td>
<td>1</td>
<td>658</td>
<td>xvbasp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>VSX Vector Absolute Double-Precision</td>
</tr>
<tr>
<td>XX2</td>
<td>1</td>
<td>658</td>
<td>xvbasp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>VSX Vector Absolute Single-Precision</td>
</tr>
<tr>
<td>XX2</td>
<td>1</td>
<td>659</td>
<td>xvaddp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>VSX Vector Add Double-Precision</td>
</tr>
<tr>
<td>XX2</td>
<td>1</td>
<td>663</td>
<td>xvadsp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>VSX Vector Add Single-Precision</td>
</tr>
<tr>
<td>XX2</td>
<td>1</td>
<td>665</td>
<td>xvcmpqep[ ]</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>VSX Vector Compare Equal Double-Precision</td>
</tr>
<tr>
<td>XX2</td>
<td>1</td>
<td>666</td>
<td>xvcmpqep[ ]</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>VSX Vector Compare Equal Single-Precision</td>
</tr>
<tr>
<td>XX2</td>
<td>1</td>
<td>667</td>
<td>xvcmpqep[ ]</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>VSX Vector Compare Greater Than or Equal Double-Precision</td>
</tr>
<tr>
<td>XX2</td>
<td>1</td>
<td>668</td>
<td>xvcmpqep[ ]</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>VSX Vector Compare Greater Than or Equal Single-Precision</td>
</tr>
<tr>
<td>XX2</td>
<td>1</td>
<td>669</td>
<td>xvcmpqdp[ ]</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>VSX Vector Compare Greater Than Double-Precision</td>
</tr>
<tr>
<td>XX2</td>
<td>1</td>
<td>670</td>
<td>xvcmpqdp[ ]</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>VSX Vector Compare Greater Than Single-Precision</td>
</tr>
<tr>
<td>XX2</td>
<td>1</td>
<td>671</td>
<td>xvcpsgnep</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>VSX Vector Copy Sign Double-Precision</td>
</tr>
<tr>
<td>XX2</td>
<td>1</td>
<td>671</td>
<td>xvcpsgnep</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>VSX Vector Copy Sign Single-Precision</td>
</tr>
<tr>
<td>XX2</td>
<td>1</td>
<td>672</td>
<td>xvcvdpdsp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>VSX Vector Convert with round Double-Precision to Single-Precision format</td>
</tr>
<tr>
<td>XX2</td>
<td>1</td>
<td>673</td>
<td>xvcvdpdps</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>VSX Vector Convert with round to zero Double-Precision to Signed Doubleword format</td>
</tr>
<tr>
<td>XX2</td>
<td>1</td>
<td>675</td>
<td>xvcvdpdpsw</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>VSX Vector Convert with round to zero Double-Precision to Signed Word format</td>
</tr>
<tr>
<td>XX2</td>
<td>1</td>
<td>677</td>
<td>xvcvdpdpsx</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>VSX Vector Convert with round to zero Double-Precision to Unsigned Doubleword format</td>
</tr>
<tr>
<td>XX2</td>
<td>1</td>
<td>679</td>
<td>xvcvdpdpsxw</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>VSX Vector Convert with round to zero Double-Precision to Unsigned Word format</td>
</tr>
<tr>
<td>XX2</td>
<td>1</td>
<td>682</td>
<td>xvcvdpdp</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>VSX Vector Convert Single-Precision to Double-Precision format</td>
</tr>
<tr>
<td>XX2</td>
<td>1</td>
<td>684</td>
<td>xvcvdpdps</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>VSX Vector Convert with round to zero Single-Precision to Signed Doubleword format</td>
</tr>
<tr>
<td>XX2</td>
<td>1</td>
<td>686</td>
<td>xvcvdpdpsx</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>VSX Vector Convert with round to zero Single-Precision to Signed Word format</td>
</tr>
</tbody>
</table>

Figure 89. Power ISA AS Instruction Set Sorted by Version (Sheet 8 of 18)
<table>
<thead>
<tr>
<th>Instruction(^1)</th>
<th>Format</th>
<th>Book</th>
<th>Page</th>
<th>Mnemonic</th>
<th>Version (^2)</th>
<th>Privilege(^3)</th>
<th>Mode Data(^4)</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>011100 ...</td>
<td></td>
<td></td>
<td></td>
<td>11000 11000.</td>
<td>XX2</td>
<td>688</td>
<td>xvcspuxds</td>
<td>v2.06</td>
</tr>
<tr>
<td>011100 ...</td>
<td></td>
<td></td>
<td></td>
<td>11000 11000.</td>
<td>XX2</td>
<td>690</td>
<td>xvcspuxws</td>
<td>v2.06</td>
</tr>
<tr>
<td>011100 ...</td>
<td></td>
<td></td>
<td></td>
<td>11111 11000.</td>
<td>XX2</td>
<td>692</td>
<td>xvcvssdxd</td>
<td>v2.06</td>
</tr>
<tr>
<td>011100 ...</td>
<td></td>
<td></td>
<td></td>
<td>11011 11000.</td>
<td>XX2</td>
<td>692</td>
<td>xvcvsxdsp</td>
<td>v2.06</td>
</tr>
<tr>
<td>011100 ...</td>
<td></td>
<td></td>
<td></td>
<td>11111 11100.</td>
<td>XX2</td>
<td>693</td>
<td>xvcsvswdps</td>
<td>v2.06</td>
</tr>
<tr>
<td>111100 ...</td>
<td></td>
<td></td>
<td></td>
<td>11000 11000.</td>
<td>XX2</td>
<td>693</td>
<td>xvcsvswsp</td>
<td>v2.06</td>
</tr>
<tr>
<td>111100 ...</td>
<td></td>
<td></td>
<td></td>
<td>11110 11000.</td>
<td>XX2</td>
<td>694</td>
<td>xvcvuxddp</td>
<td>v2.06</td>
</tr>
<tr>
<td>111100 ...</td>
<td></td>
<td></td>
<td></td>
<td>11010 11100.</td>
<td>XX2</td>
<td>694</td>
<td>xvcvuxdpsp</td>
<td>v2.06</td>
</tr>
<tr>
<td>011100 ...</td>
<td></td>
<td></td>
<td></td>
<td>11010 11100.</td>
<td>XX2</td>
<td>695</td>
<td>xvcvuxwp</td>
<td>v2.06</td>
</tr>
<tr>
<td>111100 ...</td>
<td></td>
<td></td>
<td></td>
<td>11110 11100.</td>
<td>XX2</td>
<td>695</td>
<td>xvcvuxwsp</td>
<td>v2.06</td>
</tr>
<tr>
<td>011100 ...</td>
<td></td>
<td></td>
<td></td>
<td>11111 11100.</td>
<td>XX3</td>
<td>696</td>
<td>xdvdwp</td>
<td>v2.06</td>
</tr>
<tr>
<td>011100 ...</td>
<td></td>
<td></td>
<td></td>
<td>11110 11100.</td>
<td>XX3</td>
<td>696</td>
<td>xdvdwp</td>
<td>v2.06</td>
</tr>
<tr>
<td>011100 ...</td>
<td></td>
<td></td>
<td></td>
<td>11110 11100.</td>
<td>XX3</td>
<td>701</td>
<td>xvmaddadp</td>
<td>v2.06</td>
</tr>
<tr>
<td>011100 ...</td>
<td></td>
<td></td>
<td></td>
<td>11110 11100.</td>
<td>XX3</td>
<td>704</td>
<td>xvmaddasp</td>
<td>v2.06</td>
</tr>
<tr>
<td>011100 ...</td>
<td></td>
<td></td>
<td></td>
<td>11110 11100.</td>
<td>XX3</td>
<td>701</td>
<td>xvmaddadp</td>
<td>v2.06</td>
</tr>
<tr>
<td>011100 ...</td>
<td></td>
<td></td>
<td></td>
<td>11110 11100.</td>
<td>XX3</td>
<td>704</td>
<td>xvmaddasp</td>
<td>v2.06</td>
</tr>
<tr>
<td>011100 ...</td>
<td></td>
<td></td>
<td></td>
<td>11110 11100.</td>
<td>XX3</td>
<td>707</td>
<td>xvmaddsp</td>
<td>v2.06</td>
</tr>
<tr>
<td>011100 ...</td>
<td></td>
<td></td>
<td></td>
<td>11110 11100.</td>
<td>XX3</td>
<td>709</td>
<td>xvmaxsp</td>
<td>v2.06</td>
</tr>
<tr>
<td>011100 ...</td>
<td></td>
<td></td>
<td></td>
<td>11110 11100.</td>
<td>XX3</td>
<td>711</td>
<td>xvmnsp</td>
<td>v2.06</td>
</tr>
<tr>
<td>011100 ...</td>
<td></td>
<td></td>
<td></td>
<td>11110 11100.</td>
<td>XX3</td>
<td>713</td>
<td>xvmnsp</td>
<td>v2.06</td>
</tr>
<tr>
<td>011100 ...</td>
<td></td>
<td></td>
<td></td>
<td>11110 11100.</td>
<td>XX3</td>
<td>715</td>
<td>xvmsubadp</td>
<td>v2.06</td>
</tr>
<tr>
<td>011100 ...</td>
<td></td>
<td></td>
<td></td>
<td>11110 11100.</td>
<td>XX3</td>
<td>716</td>
<td>xvmsubasp</td>
<td>v2.06</td>
</tr>
<tr>
<td>011100 ...</td>
<td></td>
<td></td>
<td></td>
<td>11110 11100.</td>
<td>XX3</td>
<td>715</td>
<td>xvmsubadp</td>
<td>v2.06</td>
</tr>
<tr>
<td>011100 ...</td>
<td></td>
<td></td>
<td></td>
<td>11110 11100.</td>
<td>XX3</td>
<td>716</td>
<td>xvmsubasp</td>
<td>v2.06</td>
</tr>
<tr>
<td>011100 ...</td>
<td></td>
<td></td>
<td></td>
<td>11110 11100.</td>
<td>XX3</td>
<td>721</td>
<td>xvmult</td>
<td>v2.06</td>
</tr>
<tr>
<td>011100 ...</td>
<td></td>
<td></td>
<td></td>
<td>11110 11100.</td>
<td>XX3</td>
<td>723</td>
<td>xvmult</td>
<td>v2.06</td>
</tr>
<tr>
<td>011100 ...</td>
<td></td>
<td></td>
<td></td>
<td>11110 11100.</td>
<td>XX2</td>
<td>725</td>
<td>xvmnabsdp</td>
<td>v2.06</td>
</tr>
<tr>
<td>011100 ...</td>
<td></td>
<td></td>
<td></td>
<td>11110 11100.</td>
<td>XX2</td>
<td>725</td>
<td>xvmnabsdp</td>
<td>v2.06</td>
</tr>
<tr>
<td>011100 ...</td>
<td></td>
<td></td>
<td></td>
<td>11110 11100.</td>
<td>XX2</td>
<td>726</td>
<td>xvmegdp</td>
<td>v2.06</td>
</tr>
<tr>
<td>011100 ...</td>
<td></td>
<td></td>
<td></td>
<td>11110 11100.</td>
<td>XX2</td>
<td>726</td>
<td>xvmegdp</td>
<td>v2.06</td>
</tr>
<tr>
<td>011100 ...</td>
<td></td>
<td></td>
<td></td>
<td>11110 11100.</td>
<td>XX3</td>
<td>727</td>
<td>xvmaddadp</td>
<td>v2.06</td>
</tr>
<tr>
<td>011100 ...</td>
<td></td>
<td></td>
<td></td>
<td>11110 11100.</td>
<td>XX3</td>
<td>732</td>
<td>xvmaddadsp</td>
<td>v2.06</td>
</tr>
<tr>
<td>011100 ...</td>
<td></td>
<td></td>
<td></td>
<td>11110 11100.</td>
<td>XX3</td>
<td>727</td>
<td>xvmaddadp</td>
<td>v2.06</td>
</tr>
<tr>
<td>011100 ...</td>
<td></td>
<td></td>
<td></td>
<td>11110 11100.</td>
<td>XX3</td>
<td>732</td>
<td>xvmaddadsp</td>
<td>v2.06</td>
</tr>
<tr>
<td>011100 ...</td>
<td></td>
<td></td>
<td></td>
<td>11110 11100.</td>
<td>XX3</td>
<td>735</td>
<td>xvmsubadp</td>
<td>v2.06</td>
</tr>
<tr>
<td>011100 ...</td>
<td></td>
<td></td>
<td></td>
<td>11110 11100.</td>
<td>XX3</td>
<td>738</td>
<td>xvmsubadsp</td>
<td>v2.06</td>
</tr>
<tr>
<td>011100 ...</td>
<td></td>
<td></td>
<td></td>
<td>11110 11100.</td>
<td>XX3</td>
<td>735</td>
<td>xvmsubadp</td>
<td>v2.06</td>
</tr>
<tr>
<td>011100 ...</td>
<td></td>
<td></td>
<td></td>
<td>11110 11100.</td>
<td>XX3</td>
<td>738</td>
<td>xvmsubadsp</td>
<td>v2.06</td>
</tr>
<tr>
<td>011100 ...</td>
<td></td>
<td></td>
<td></td>
<td>11110 11100.</td>
<td>XX2</td>
<td>741</td>
<td>xvdpsi</td>
<td>v2.06</td>
</tr>
<tr>
<td>011100 ...</td>
<td></td>
<td></td>
<td></td>
<td>11111 11110.</td>
<td>XX2</td>
<td>741</td>
<td>xvdpsi</td>
<td>v2.06</td>
</tr>
<tr>
<td>011100 ...</td>
<td></td>
<td></td>
<td></td>
<td>11111 11110.</td>
<td>XX2</td>
<td>742</td>
<td>xvdpsip</td>
<td>v2.06</td>
</tr>
<tr>
<td>011100 ...</td>
<td></td>
<td></td>
<td></td>
<td>11111 11110.</td>
<td>XX2</td>
<td>742</td>
<td>xvdpsip</td>
<td>v2.06</td>
</tr>
<tr>
<td>011100 ...</td>
<td></td>
<td></td>
<td></td>
<td>11111 11110.</td>
<td>XX2</td>
<td>743</td>
<td>xvdpsip</td>
<td>v2.06</td>
</tr>
<tr>
<td>011100 ...</td>
<td></td>
<td></td>
<td></td>
<td>11111 11110.</td>
<td>XX2</td>
<td>744</td>
<td>xvdpsip</td>
<td>v2.06</td>
</tr>
<tr>
<td>011100 ...</td>
<td></td>
<td></td>
<td></td>
<td>11111 11110.</td>
<td>XX2</td>
<td>745</td>
<td>xvdpsip</td>
<td>v2.06</td>
</tr>
<tr>
<td>011100 ...</td>
<td></td>
<td></td>
<td></td>
<td>11111 11110.</td>
<td>XX2</td>
<td>746</td>
<td>xvdpsip</td>
<td>v2.06</td>
</tr>
<tr>
<td>011100 ...</td>
<td></td>
<td></td>
<td></td>
<td>11111 11110.</td>
<td>XX2</td>
<td>746</td>
<td>xvdpsip</td>
<td>v2.06</td>
</tr>
<tr>
<td>011100 ...</td>
<td></td>
<td></td>
<td></td>
<td>11111 11110.</td>
<td>XX2</td>
<td>747</td>
<td>xvdpsip</td>
<td>v2.06</td>
</tr>
</tbody>
</table>

Figure 89. Power ISA AS Instruction Set Sorted by Version (Sheet 9 of 18)
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>Book</th>
<th>Page</th>
<th>Mnemonic</th>
<th>Version</th>
<th>Privilege</th>
<th>Mode</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>XX2</td>
<td>747</td>
<td>v2.06</td>
<td>v2.06</td>
<td>VSX Vector Round Single-Precision to Integral toward +Infinity</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>XX2</td>
<td>748</td>
<td>v2.06</td>
<td>v2.06</td>
<td>VSX Vector Round Single-Precision to Integral toward Zero</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>XX2</td>
<td>748</td>
<td>v2.06</td>
<td>v2.06</td>
<td>VSX Vector Reciprocal Square Root Estimate Double-Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>XX2</td>
<td>750</td>
<td>v2.06</td>
<td>v2.06</td>
<td>VSX Vector Reciprocal Square Root Estimate Single-Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>XX2</td>
<td>751</td>
<td>v2.06</td>
<td>v2.06</td>
<td>VSX Vector Square Root Double-Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>XX2</td>
<td>752</td>
<td>v2.06</td>
<td>v2.06</td>
<td>VSX Vector Square Root Single-Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>XX2</td>
<td>753</td>
<td>v2.06</td>
<td>v2.06</td>
<td>VSX Vector Subtract Double-Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>XX2</td>
<td>755</td>
<td>v2.06</td>
<td>v2.06</td>
<td>VSX Vector Subtract Single-Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>XX3</td>
<td>757</td>
<td>v2.06</td>
<td>v2.06</td>
<td>VSX Vector Test for software Divide Double-Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>XX3</td>
<td>758</td>
<td>v2.06</td>
<td>v2.06</td>
<td>VSX Vector Test for software Divide Single-Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>XX3</td>
<td>759</td>
<td>v2.06</td>
<td>v2.06</td>
<td>VSX Vector Test for software Square Root Double-Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>XX3</td>
<td>759</td>
<td>v2.06</td>
<td>v2.06</td>
<td>VSX Vector Test for software Square Root Single-Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>XX3</td>
<td>767</td>
<td>v2.06</td>
<td>v2.06</td>
<td>VSX Vector Logical AND</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>XX3</td>
<td>767</td>
<td>v2.06</td>
<td>v2.06</td>
<td>VSX Vector Logical AND with Complement</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>XX3</td>
<td>769</td>
<td>v2.06</td>
<td>v2.06</td>
<td>VSX Vector Logical OR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>XX3</td>
<td>770</td>
<td>v2.06</td>
<td>v2.06</td>
<td>VSX Vector Logical XOR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>XX3</td>
<td>771</td>
<td>v2.06</td>
<td>v2.06</td>
<td>VSX Vector Merge Word High</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>XX3</td>
<td>771</td>
<td>v2.06</td>
<td>v2.06</td>
<td>VSX Vector Merge Word Low</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>XX3</td>
<td>773</td>
<td>v2.06</td>
<td>v2.06</td>
<td>VSX Vector Doubleword Permute Immediate</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>XX3</td>
<td>774</td>
<td>v2.06</td>
<td>v2.06</td>
<td>VSX Vector Select</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>XX3</td>
<td>774</td>
<td>v2.06</td>
<td>v2.06</td>
<td>VSX Vector Shift Left Double by Word Immediate</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>XX3</td>
<td>774</td>
<td>v2.06</td>
<td>v2.06</td>
<td>VSX Vector Splat Word</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>197</td>
<td>v2.05</td>
<td>v2.05</td>
<td>Compare byte</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>193</td>
<td>v2.05</td>
<td>v2.05</td>
<td>DFP Add</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>193</td>
<td>v2.05</td>
<td>v2.05</td>
<td>DFP Add Quad</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>215</td>
<td>v2.05</td>
<td>v2.05</td>
<td>DFP Convert From Fixed Quad</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>199</td>
<td>v2.05</td>
<td>v2.05</td>
<td>DFP Compare Ordered</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>199</td>
<td>v2.05</td>
<td>v2.05</td>
<td>DFP Compare Ordered</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>198</td>
<td>v2.05</td>
<td>v2.05</td>
<td>DFP Compare Unordered</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>198</td>
<td>v2.05</td>
<td>v2.05</td>
<td>DFP Compare Unordered Quad</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>213</td>
<td>v2.05</td>
<td>v2.05</td>
<td>DFP Convert To DFP Long</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>215</td>
<td>v2.05</td>
<td>v2.05</td>
<td>DFP Convert To Fixed</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>215</td>
<td>v2.05</td>
<td>v2.05</td>
<td>DFP Convert To Fixed Quad</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>213</td>
<td>v2.05</td>
<td>v2.05</td>
<td>DFP Convert To DFP Extended</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>217</td>
<td>v2.05</td>
<td>v2.05</td>
<td>DFP Decompress DPD to BCD</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>217</td>
<td>v2.05</td>
<td>v2.05</td>
<td>DFP Decode DPD to BCD Quad</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>196</td>
<td>v2.05</td>
<td>v2.05</td>
<td>DFP Divide</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>196</td>
<td>v2.05</td>
<td>v2.05</td>
<td>DFP Divide Quad</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>217</td>
<td>v2.05</td>
<td>v2.05</td>
<td>DFP Encode BCD to DPD</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>217</td>
<td>v2.05</td>
<td>v2.05</td>
<td>DFP Encode BCD to DPD Quad</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>218</td>
<td>v2.05</td>
<td>v2.05</td>
<td>DFP Insert Exponent</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>218</td>
<td>v2.05</td>
<td>v2.05</td>
<td>DFP Insert Exponent Quad</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>195</td>
<td>v2.05</td>
<td>v2.05</td>
<td>DFP Multiply</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>195</td>
<td>v2.05</td>
<td>v2.05</td>
<td>DFP Multiply Quad</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>204</td>
<td>v2.05</td>
<td>v2.05</td>
<td>DFP Quantize</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>204</td>
<td>v2.05</td>
<td>v2.05</td>
<td>DFP Quantize Immediate</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>203</td>
<td>v2.05</td>
<td>v2.05</td>
<td>DFP Quantize Immediate Quad</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>204</td>
<td>v2.05</td>
<td>v2.05</td>
<td>DFP Quantize Quad</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>214</td>
<td>v2.05</td>
<td>v2.05</td>
<td>DFP Round To DFP Long</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>211</td>
<td>v2.05</td>
<td>v2.05</td>
<td>DFP Round To FP Integer Inexact</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>211</td>
<td>v2.05</td>
<td>v2.05</td>
<td>DFP Round To FP Integer Inexact Quad</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>209</td>
<td>v2.05</td>
<td>v2.05</td>
<td>DFP Round To FP Integer Inexact</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>209</td>
<td>v2.05</td>
<td>v2.05</td>
<td>DFP Round To FP Integer Inexact Quad</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>206</td>
<td>v2.05</td>
<td>v2.05</td>
<td>DFP Round</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>206</td>
<td>v2.05</td>
<td>v2.05</td>
<td>DFP Round Quad</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 89. Power ISA AS Instruction Set Sorted by Version (Sheet 10 of 18)
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>Book</th>
<th>Page</th>
<th>Mnemonic</th>
<th>Version 2</th>
<th>Privilege 3</th>
<th>Mode Dep 4</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>111</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>110</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>98</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>96</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>95</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>94</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>93</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>92</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>91</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>90</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>89</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>88</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>87</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>86</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>85</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>84</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>83</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Figure 89. Power ISA AS Instruction Set Sorted by Version (Sheet 11 of 18)
<table>
<thead>
<tr>
<th>Instruction1</th>
<th>Format</th>
<th>Book</th>
<th>Page</th>
<th>Mnemonic</th>
<th>Version2</th>
<th>Privilege3</th>
<th>ModeDep4</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>00100</td>
<td>VX</td>
<td>269</td>
<td>vaddshs</td>
<td>v2.03</td>
<td>Vector Add Signed Halfword Saturate</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>270</td>
<td>vaddsws</td>
<td>v2.03</td>
<td>Vector Add Signed Word Saturate</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>270</td>
<td>vaddum</td>
<td>v2.03</td>
<td>Vector Add Unsigned Byte Modulo</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>272</td>
<td>vaddubs</td>
<td>v2.03</td>
<td>Vector Add Unsigned Byte Saturate</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>271</td>
<td>vaddumh</td>
<td>v2.03</td>
<td>Vector Add Unsigned Halfword Modulo</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>271</td>
<td>vadduhb</td>
<td>v2.03</td>
<td>Vector Add Unsigned Halfword Saturate</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>271</td>
<td>vaddumw</td>
<td>v2.03</td>
<td>Vector Add Unsigned Word Modulo</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>272</td>
<td>vaddusw</td>
<td>v2.03</td>
<td>Vector Add Unsigned Word Saturate</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>272</td>
<td>vadduhw</td>
<td>v2.03</td>
<td>Vector Add Unsigned Halfword Saturate</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>312</td>
<td>vand</td>
<td>v2.03</td>
<td>Vector Logical AND</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>312</td>
<td>vandc</td>
<td>v2.03</td>
<td>Vector Logical AND with Complement</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>325</td>
<td>vcfsx</td>
<td>v2.03</td>
<td>Vector Convert with round to nearest Signed Word format to FP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>325</td>
<td>vdfux</td>
<td>v2.03</td>
<td>Vector Convert with round to nearest Unsigned Word format to FP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>328</td>
<td>vcmpbf[.]</td>
<td>v2.03</td>
<td>Vector Compare Bounds Floating-Point</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>328</td>
<td>vcmpeqbf[.]</td>
<td>v2.03</td>
<td>Vector Compare Equal To Floating-Point</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>330</td>
<td>vcmpuebf[.]</td>
<td>v2.03</td>
<td>Vector Compare Equal To Unsigned Byte</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>330</td>
<td>vcmpequebf[.]</td>
<td>v2.03</td>
<td>Vector Compare Equal To Unsigned Halfword</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>340</td>
<td>vcmpequew[.]</td>
<td>v2.03</td>
<td>Vector Compare Equal To Unsigned Word</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>329</td>
<td>vcmpgtbf[.]</td>
<td>v2.03</td>
<td>Vector Compare Greater Than or Equal To Floating-Point</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>330</td>
<td>vcmpgtbf[.]</td>
<td>v2.03</td>
<td>Vector Compare Greater Than Floating-Point</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>330</td>
<td>vcmpgtsbf[.]</td>
<td>v2.03</td>
<td>Vector Compare Greater Than Signed Byte</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>329</td>
<td>vcmpgtbtf[.]</td>
<td>v2.03</td>
<td>Vector Compare Greater Than Signed Halfword</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>330</td>
<td>vcmpequbf[.]</td>
<td>v2.03</td>
<td>Vector Compare Greater Than Signed Word</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>329</td>
<td>vcmpequtbf[.]</td>
<td>v2.03</td>
<td>Vector Compare Greater Than Unsigned Byte</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>328</td>
<td>vcmpequtf[.]</td>
<td>v2.03</td>
<td>Vector Compare Greater Than Unsigned Halfword</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>327</td>
<td>vcmpequtfw[.]</td>
<td>v2.03</td>
<td>Vector Compare Greater Than Unsigned Word</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>324</td>
<td>vdfxs</td>
<td>v2.03</td>
<td>Vector Convert with round to zero FP To Signed Word format Saturate</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>324</td>
<td>vdfuxs</td>
<td>v2.03</td>
<td>Vector Convert with round to zero FP To Unsigned Word format Saturate</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>331</td>
<td>vextelpf[.]</td>
<td>v2.03</td>
<td>Vector Log Base 2 Estimate Floating-Point</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>331</td>
<td>vlogelpf[.]</td>
<td>v2.03</td>
<td>Vector Log Base 2 Estimate Floating-Point</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00100</td>
<td>VA</td>
<td>322</td>
<td>vmadfp[.]</td>
<td>v2.03</td>
<td>Vector Multiply-Add Floating-Point</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>323</td>
<td>vmaxdp[.]</td>
<td>v2.03</td>
<td>Vector Maximum Floating-Point</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>329</td>
<td>vmaxsb[.]</td>
<td>v2.03</td>
<td>Vector Maximum Signed Byte</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>330</td>
<td>vmaxsh[.]</td>
<td>v2.03</td>
<td>Vector Maximum Signed Halfword</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>330</td>
<td>vmaxsw[.]</td>
<td>v2.03</td>
<td>Vector Maximum Signed Word</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>329</td>
<td>vmaxub[.]</td>
<td>v2.03</td>
<td>Vector Maximum Unsigned Byte</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>330</td>
<td>vmaxuh[.]</td>
<td>v2.03</td>
<td>Vector Maximum Unsigned Halfword</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>330</td>
<td>vmaxuw[.]</td>
<td>v2.03</td>
<td>Vector Maximum Unsigned Word</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00100</td>
<td>VA</td>
<td>285</td>
<td>vmraddshs</td>
<td>v2.03</td>
<td>Vector Multiply-High-Add Signed Halfword Saturate</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00100</td>
<td>VA</td>
<td>285</td>
<td>vnmraddshs</td>
<td>v2.03</td>
<td>Vector Multiply-High-Round-Add Signed Halfword Saturate</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>323</td>
<td>vnmf[.]</td>
<td>v2.03</td>
<td>Vector Minimum Floating-Point</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>301</td>
<td>vmins[.]</td>
<td>v2.03</td>
<td>Vector Minimum Signed Byte</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>302</td>
<td>vminsh[.]</td>
<td>v2.03</td>
<td>Vector Minimum Signed Halfword</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>302</td>
<td>vmins[.]</td>
<td>v2.03</td>
<td>Vector Minimum Signed Word</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>301</td>
<td>vminub[.]</td>
<td>v2.03</td>
<td>Vector Minimum Unsigned Byte</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>302</td>
<td>vminuh[.]</td>
<td>v2.03</td>
<td>Vector Minimum Unsigned Halfword</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>302</td>
<td>vminuw[.]</td>
<td>v2.03</td>
<td>Vector Minimum Unsigned Word</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00100</td>
<td>VA</td>
<td>286</td>
<td>vmladduhm</td>
<td>v2.03</td>
<td>Vector Multiply-Low-Add Unsigned Halfword Modulo</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>255</td>
<td>vmrghb[.]</td>
<td>v2.03</td>
<td>Vector Merge High Byte</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 89. Power ISA AS Instruction Set Sorted by Version (Sheet 12 of 18)

1210 Power ISA™ Appendices
<table>
<thead>
<tr>
<th>Instruction&lt;sup&gt;1&lt;/sup&gt;</th>
<th>Format</th>
<th>Book</th>
<th>Page</th>
<th>Mnemonic</th>
<th>Version&lt;sup&gt;2&lt;/sup&gt;</th>
<th>Privilege&lt;sup&gt;3&lt;/sup&gt;</th>
<th>Mode Dep&lt;sup&gt;4&lt;/sup&gt;</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0010</td>
<td>VA</td>
<td></td>
<td>1</td>
<td>vx</td>
<td>255 vrnghh</td>
<td>v2.03</td>
<td></td>
<td>Vector Merge High Halfword</td>
</tr>
<tr>
<td>0x0010</td>
<td>VX</td>
<td></td>
<td>1</td>
<td>vx</td>
<td>256 vrnghw</td>
<td>v2.03</td>
<td></td>
<td>Vector Merge High Word</td>
</tr>
<tr>
<td>0x0010</td>
<td>VX</td>
<td></td>
<td>1</td>
<td>vx</td>
<td>255 vrngb</td>
<td>v2.03</td>
<td></td>
<td>Vector Merge Low Byte</td>
</tr>
<tr>
<td>0x0010</td>
<td>VX</td>
<td></td>
<td>1</td>
<td>vx</td>
<td>255 vrnghf</td>
<td>v2.03</td>
<td></td>
<td>Vector Merge Low Halfword</td>
</tr>
<tr>
<td>0x0010</td>
<td>VX</td>
<td></td>
<td>1</td>
<td>vx</td>
<td>256 vrnghw</td>
<td>v2.03</td>
<td></td>
<td>Vector Merge Low Word</td>
</tr>
<tr>
<td>0x1010</td>
<td>VA</td>
<td></td>
<td>1</td>
<td>va</td>
<td>287 vmsummbm</td>
<td>v2.03</td>
<td></td>
<td>Vector Multiply-Sum Mixed Byte Modulo</td>
</tr>
<tr>
<td>0x1010</td>
<td>VA</td>
<td></td>
<td>1</td>
<td>va</td>
<td>287 vmsumshm</td>
<td>v2.03</td>
<td></td>
<td>Vector Multiply-Sum Signed Halfword Modulo</td>
</tr>
<tr>
<td>0x1010</td>
<td>VA</td>
<td></td>
<td>1</td>
<td>va</td>
<td>286 vmsumshs</td>
<td>v2.03</td>
<td></td>
<td>Vector Multiply-Sum Signed Halfword Saturate</td>
</tr>
<tr>
<td>0x1010</td>
<td>VA</td>
<td></td>
<td>1</td>
<td>va</td>
<td>286 vmsumsbm</td>
<td>v2.03</td>
<td></td>
<td>Vector Multiply-Sum Unsigned Byte Modulo</td>
</tr>
<tr>
<td>0x1010</td>
<td>VA</td>
<td></td>
<td>1</td>
<td>va</td>
<td>288 vmsumuhm</td>
<td>v2.03</td>
<td></td>
<td>Vector Multiply-Sum Unsigned Halfword Modulo</td>
</tr>
<tr>
<td>0x1010</td>
<td>VA</td>
<td></td>
<td>1</td>
<td>va</td>
<td>289 vmsumuh</td>
<td>v2.03</td>
<td></td>
<td>Vector Multiply-Sum Unsigned Halfword Saturate</td>
</tr>
<tr>
<td>0x1010</td>
<td>VX</td>
<td></td>
<td>1</td>
<td>vx</td>
<td>281 vmsubslb</td>
<td>v2.03</td>
<td></td>
<td>Vector Multiply Even Signed Byte</td>
</tr>
<tr>
<td>0x1010</td>
<td>VX</td>
<td></td>
<td>1</td>
<td>vx</td>
<td>282 vmsubsh</td>
<td>v2.03</td>
<td></td>
<td>Vector Multiply Even Signed Halfword</td>
</tr>
<tr>
<td>0x1010</td>
<td>VX</td>
<td></td>
<td>1</td>
<td>vx</td>
<td>281 vmsubslb</td>
<td>v2.03</td>
<td></td>
<td>Vector Multiply Even Unsigned Byte</td>
</tr>
<tr>
<td>0x1010</td>
<td>VX</td>
<td></td>
<td>1</td>
<td>vx</td>
<td>282 vmsubsh</td>
<td>v2.03</td>
<td></td>
<td>Vector Multiply Even Unsigned Halfword</td>
</tr>
<tr>
<td>0x1010</td>
<td>VX</td>
<td></td>
<td>1</td>
<td>vx</td>
<td>281 vmsubslb</td>
<td>v2.03</td>
<td></td>
<td>Vector Multiply Odd Signed Byte</td>
</tr>
<tr>
<td>0x1010</td>
<td>VX</td>
<td></td>
<td>1</td>
<td>vx</td>
<td>282 vmsubsh</td>
<td>v2.03</td>
<td></td>
<td>Vector Multiply Odd Signed Halfword</td>
</tr>
<tr>
<td>0x1010</td>
<td>VX</td>
<td></td>
<td>1</td>
<td>vx</td>
<td>282 vmsubsh</td>
<td>v2.03</td>
<td></td>
<td>Vector Multiply Odd Unsigned Halfword</td>
</tr>
<tr>
<td>0x1010</td>
<td>VA</td>
<td></td>
<td>1</td>
<td>va</td>
<td>322 vmsubdplp</td>
<td>v2.03</td>
<td></td>
<td>Vector Negative Multiply-Subtract Floating-Point</td>
</tr>
<tr>
<td>0x1010</td>
<td>VX</td>
<td></td>
<td>1</td>
<td>vx</td>
<td>313 vnior</td>
<td>v2.03</td>
<td></td>
<td>Vector Logical NOR</td>
</tr>
<tr>
<td>0x1010</td>
<td>VX</td>
<td></td>
<td>1</td>
<td>vx</td>
<td>313 vior</td>
<td>v2.03</td>
<td></td>
<td>Vector Logical OR</td>
</tr>
<tr>
<td>0x1010</td>
<td>VA</td>
<td></td>
<td>1</td>
<td>va</td>
<td>320 vperm</td>
<td>v2.03</td>
<td></td>
<td>Vector Permute</td>
</tr>
<tr>
<td>0x1010</td>
<td>VX</td>
<td></td>
<td>1</td>
<td>vx</td>
<td>248 vpkpix</td>
<td>v2.03</td>
<td></td>
<td>Vector Pack Pixel</td>
</tr>
<tr>
<td>0x1010</td>
<td>VX</td>
<td></td>
<td>1</td>
<td>vx</td>
<td>249 vpkshiss</td>
<td>v2.03</td>
<td></td>
<td>Vector Pack Signed Halfword Saturated</td>
</tr>
<tr>
<td>0x1010</td>
<td>VX</td>
<td></td>
<td>1</td>
<td>vx</td>
<td>250 vpkshius</td>
<td>v2.03</td>
<td></td>
<td>Vector Pack Signed Halfword Unsaturated</td>
</tr>
<tr>
<td>0x1010</td>
<td>VX</td>
<td></td>
<td>1</td>
<td>vx</td>
<td>250 vpkswss</td>
<td>v2.03</td>
<td></td>
<td>Vector Pack Signed Word Saturated</td>
</tr>
<tr>
<td>0x1010</td>
<td>VX</td>
<td></td>
<td>1</td>
<td>vx</td>
<td>251 vpkswus</td>
<td>v2.03</td>
<td></td>
<td>Vector Pack Signed Word Unsaturated</td>
</tr>
<tr>
<td>0x1010</td>
<td>VX</td>
<td></td>
<td>1</td>
<td>vx</td>
<td>251 vpkshum</td>
<td>v2.03</td>
<td></td>
<td>Vector Pack Signed Halfword Unsaturated Modulo</td>
</tr>
<tr>
<td>0x1010</td>
<td>VX</td>
<td></td>
<td>1</td>
<td>vx</td>
<td>252 vpkshus</td>
<td>v2.03</td>
<td></td>
<td>Vector Pack Signed Halfword Unsaturated Saturate</td>
</tr>
<tr>
<td>0x1010</td>
<td>VX</td>
<td></td>
<td>1</td>
<td>vx</td>
<td>252 vpkshum</td>
<td>v2.03</td>
<td></td>
<td>Vector Pack Signed Word Unsaturated Modulo</td>
</tr>
<tr>
<td>0x1010</td>
<td>VX</td>
<td></td>
<td>1</td>
<td>vx</td>
<td>252 vpkshus</td>
<td>v2.03</td>
<td></td>
<td>Vector Pack Signed Word Unsaturated Saturate</td>
</tr>
<tr>
<td>0x1010</td>
<td>VX</td>
<td></td>
<td>1</td>
<td>vx</td>
<td>332 vrepf</td>
<td>v2.03</td>
<td></td>
<td>Vector Reciprocal Estimate Floating-Point</td>
</tr>
<tr>
<td>0x1010</td>
<td>VX</td>
<td></td>
<td>1</td>
<td>vx</td>
<td>326 vrfhi</td>
<td>v2.03</td>
<td></td>
<td>Vector Round to Floating-Point Integral toward -Infinity</td>
</tr>
<tr>
<td>0x1010</td>
<td>VX</td>
<td></td>
<td>1</td>
<td>vx</td>
<td>326 vrfhi</td>
<td>v2.03</td>
<td></td>
<td>Vector Round to Floating-Point Integral Nearest</td>
</tr>
<tr>
<td>0x1010</td>
<td>VX</td>
<td></td>
<td>1</td>
<td>vx</td>
<td>326 vrfhip</td>
<td>v2.03</td>
<td></td>
<td>Vector Round to Floating-Point Integral toward +Infinity</td>
</tr>
<tr>
<td>0x1010</td>
<td>VX</td>
<td></td>
<td>1</td>
<td>vx</td>
<td>327 vrflc</td>
<td>v2.03</td>
<td></td>
<td>Vector Round to Floating-Point Integral toward Zero</td>
</tr>
<tr>
<td>0x1010</td>
<td>VX</td>
<td></td>
<td>1</td>
<td>vx</td>
<td>315 vrtch</td>
<td>v2.03</td>
<td></td>
<td>Vector Rotate Left Byte</td>
</tr>
<tr>
<td>0x1010</td>
<td>VX</td>
<td></td>
<td>1</td>
<td>vx</td>
<td>315 vrth</td>
<td>v2.03</td>
<td></td>
<td>Vector Rotate Left Halfword</td>
</tr>
<tr>
<td>0x1010</td>
<td>VX</td>
<td></td>
<td>1</td>
<td>vx</td>
<td>315 vrtc</td>
<td>v2.03</td>
<td></td>
<td>Vector Rotate Left Word</td>
</tr>
<tr>
<td>0x1010</td>
<td>VX</td>
<td></td>
<td>1</td>
<td>vx</td>
<td>332 vrsqrtfp</td>
<td>v2.03</td>
<td></td>
<td>Vector Reciprocal Square Root Estimate Floating-Point</td>
</tr>
<tr>
<td>0x1010</td>
<td>VA</td>
<td></td>
<td>1</td>
<td>va</td>
<td>261 vsel</td>
<td>v2.03</td>
<td></td>
<td>Vector Select</td>
</tr>
<tr>
<td>0x1010</td>
<td>VX</td>
<td></td>
<td>1</td>
<td>vx</td>
<td>264 vsl</td>
<td>v2.03</td>
<td></td>
<td>Vector Shift Left</td>
</tr>
<tr>
<td>0x1010</td>
<td>VX</td>
<td></td>
<td>1</td>
<td>vx</td>
<td>316 vsib</td>
<td>v2.03</td>
<td></td>
<td>Vector Shift Left Byte</td>
</tr>
<tr>
<td>0x1010</td>
<td>VA</td>
<td></td>
<td>1</td>
<td>va</td>
<td>263 vsdci</td>
<td>v2.03</td>
<td></td>
<td>Vector Shift Left Double by Octet Immediate</td>
</tr>
<tr>
<td>0x1010</td>
<td>VX</td>
<td></td>
<td>1</td>
<td>vx</td>
<td>316 vsih</td>
<td>v2.03</td>
<td></td>
<td>Vector Shift Left Halfword</td>
</tr>
<tr>
<td>0x1010</td>
<td>VX</td>
<td></td>
<td>1</td>
<td>vx</td>
<td>264 vsio</td>
<td>v2.03</td>
<td></td>
<td>Vector Shift Left by Octet</td>
</tr>
<tr>
<td>0x1010</td>
<td>VX</td>
<td></td>
<td>1</td>
<td>vx</td>
<td>316 vsiw</td>
<td>v2.03</td>
<td></td>
<td>Vector Shift Left Word</td>
</tr>
<tr>
<td>0x1010</td>
<td>VX</td>
<td></td>
<td>1</td>
<td>vx</td>
<td>258 vsipib</td>
<td>v2.03</td>
<td></td>
<td>Vector Splat Byte</td>
</tr>
<tr>
<td>0x1010</td>
<td>VX</td>
<td></td>
<td>1</td>
<td>vx</td>
<td>258 vsipif</td>
<td>v2.03</td>
<td></td>
<td>Vector Splat Halfword</td>
</tr>
<tr>
<td>0x1010</td>
<td>VX</td>
<td></td>
<td>1</td>
<td>vx</td>
<td>259 vsiplab</td>
<td>v2.03</td>
<td></td>
<td>Vector Splat Immediate Signed Byte</td>
</tr>
<tr>
<td>0x1010</td>
<td>VX</td>
<td></td>
<td>1</td>
<td>vx</td>
<td>259 vsipish</td>
<td>v2.03</td>
<td></td>
<td>Vector Splat Immediate Signed Halfword</td>
</tr>
<tr>
<td>0x1010</td>
<td>VX</td>
<td></td>
<td>1</td>
<td>vx</td>
<td>259 vsipisw</td>
<td>v2.03</td>
<td></td>
<td>Vector Splat Immediate Signed Word</td>
</tr>
<tr>
<td>0x1010</td>
<td>VX</td>
<td></td>
<td>1</td>
<td>vx</td>
<td>258 vsiplw</td>
<td>v2.03</td>
<td></td>
<td>Vector Splat Word</td>
</tr>
<tr>
<td>0x1010</td>
<td>VX</td>
<td></td>
<td>1</td>
<td>vx</td>
<td>264 vsl</td>
<td>v2.03</td>
<td></td>
<td>Vector Shift Right</td>
</tr>
<tr>
<td>0x1010</td>
<td>VX</td>
<td></td>
<td>1</td>
<td>vx</td>
<td>318 vsrabc</td>
<td>v2.03</td>
<td></td>
<td>Vector Shift Right Algebraic Byte</td>
</tr>
</tbody>
</table>

**Figure 89. Power ISA AS Instruction Set Sorted by Version (Sheet 13 of 18)**
<table>
<thead>
<tr>
<th>Instruction¹</th>
<th>Format</th>
<th>Book</th>
<th>Page</th>
<th>Mnemonic</th>
<th>Version²</th>
<th>Privilege³</th>
<th>Mode Dep⁴</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>001000</td>
<td>VX</td>
<td>318</td>
<td></td>
<td>vsrah</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>Vector Shift Right Algebraic Halfword</td>
</tr>
<tr>
<td>001000</td>
<td>VX</td>
<td>318</td>
<td></td>
<td>vsraw</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>Vector Shift Right Algebraic Word</td>
</tr>
<tr>
<td>001000</td>
<td>VX</td>
<td>317</td>
<td></td>
<td>vsrb</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>Vector Shift Right Byte</td>
</tr>
<tr>
<td>001000</td>
<td>VX</td>
<td>317</td>
<td></td>
<td>vsrh</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>Vector Shift Right Halfword</td>
</tr>
<tr>
<td>001000</td>
<td>VX</td>
<td>264</td>
<td></td>
<td>vsro</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>Vector Shift Right by Octet</td>
</tr>
<tr>
<td>001000</td>
<td>VX</td>
<td>317</td>
<td></td>
<td>vsrw</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>Vector Shift Right Word</td>
</tr>
<tr>
<td>010100</td>
<td>VX</td>
<td>275</td>
<td></td>
<td>vsubcw</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>Vector Subtract &amp; Write-Carry-Out Unsigned Word</td>
</tr>
<tr>
<td>001000</td>
<td>VX</td>
<td>321</td>
<td></td>
<td>vsubfb</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>Vector Subtract Floating-Point</td>
</tr>
<tr>
<td>011000</td>
<td>VX</td>
<td>275</td>
<td></td>
<td>vsubhs</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>Vector Subtract Signed Halfword</td>
</tr>
<tr>
<td>001000</td>
<td>VX</td>
<td>276</td>
<td></td>
<td>vsubsws</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>Vector Subtract Signed Word</td>
</tr>
<tr>
<td>010000</td>
<td>VX</td>
<td>277</td>
<td></td>
<td>vsubwds</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>Vector Subtract Signed Word Module</td>
</tr>
<tr>
<td>010000</td>
<td>VX</td>
<td>278</td>
<td></td>
<td>vsububs</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>Vector Subtract Unsigned Byte</td>
</tr>
<tr>
<td>010000</td>
<td>VX</td>
<td>277</td>
<td></td>
<td>vsubuhm</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>Vector Subtract Unsigned Halfword Modulo</td>
</tr>
<tr>
<td>010000</td>
<td>VX</td>
<td>278</td>
<td></td>
<td>vsubuhm</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>Vector Subtract Unsigned Halfword</td>
</tr>
<tr>
<td>010000</td>
<td>VX</td>
<td>277</td>
<td></td>
<td>vsubuhw</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>Vector Subtract Unsigned Word Module</td>
</tr>
<tr>
<td>010000</td>
<td>VX</td>
<td>278</td>
<td></td>
<td>vsubuws</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>Vector Subtract Unsigned Word</td>
</tr>
<tr>
<td>111000</td>
<td>VX</td>
<td>290</td>
<td></td>
<td>vsms2ws</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>Vector Sum across Half Signed Word Saturate</td>
</tr>
<tr>
<td>011000</td>
<td>VX</td>
<td>291</td>
<td></td>
<td>vsmsksbs</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>Vector Sum across Quarter Signed Byte Saturate</td>
</tr>
<tr>
<td>011000</td>
<td>VX</td>
<td>291</td>
<td></td>
<td>vsmsksht</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>Vector Sum across Quarter Signed Halfword Saturate</td>
</tr>
<tr>
<td>011000</td>
<td>VX</td>
<td>292</td>
<td></td>
<td>vsmskhs</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>Vector Sum across Quarter Signed Byte</td>
</tr>
<tr>
<td>011000</td>
<td>VX</td>
<td>290</td>
<td></td>
<td>vsmsws</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>Vector Sum across Signed Word Saturate</td>
</tr>
<tr>
<td>011000</td>
<td>VX</td>
<td>254</td>
<td></td>
<td>vukphs</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>Vector Unpack High Signed Byte</td>
</tr>
<tr>
<td>011000</td>
<td>VX</td>
<td>254</td>
<td></td>
<td>vukphsh</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>Vector Unpack High Signed Halfword</td>
</tr>
<tr>
<td>011000</td>
<td>VX</td>
<td>253</td>
<td></td>
<td>vukphpx</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>Vector Unpack High Signed Byte</td>
</tr>
<tr>
<td>011000</td>
<td>VX</td>
<td>253</td>
<td></td>
<td>vukphps</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>Vector Unpack High Signed Halfword</td>
</tr>
<tr>
<td>011000</td>
<td>VX</td>
<td>254</td>
<td></td>
<td>vukphs</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>Vector Unpack Low Pixel</td>
</tr>
<tr>
<td>011000</td>
<td>VX</td>
<td>254</td>
<td></td>
<td>vukphkb</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>Vector Unpack Low Signed Byte</td>
</tr>
<tr>
<td>011000</td>
<td>VX</td>
<td>254</td>
<td></td>
<td>vukphks</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>Vector Unpack Low Signed Halfword</td>
</tr>
<tr>
<td>111000</td>
<td>VX</td>
<td>313</td>
<td></td>
<td>vxor</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>Vector Logical XOR</td>
</tr>
<tr>
<td>111000</td>
<td>VX</td>
<td>154</td>
<td></td>
<td>fre[.a]</td>
<td>v2.02</td>
<td></td>
<td></td>
<td>Floating Reciprocal Estimate</td>
</tr>
<tr>
<td>111000</td>
<td>VX</td>
<td>166</td>
<td></td>
<td>frn[.a]</td>
<td>v2.02</td>
<td></td>
<td></td>
<td>Floating Round To Integer Minus</td>
</tr>
<tr>
<td>111000</td>
<td>VX</td>
<td>166</td>
<td></td>
<td>frn[.a]</td>
<td>v2.02</td>
<td></td>
<td></td>
<td>Floating Round To Integer Nearest</td>
</tr>
<tr>
<td>111000</td>
<td>VX</td>
<td>166</td>
<td></td>
<td>frp[.a]</td>
<td>v2.02</td>
<td></td>
<td></td>
<td>Floating Round To Integer Plus</td>
</tr>
<tr>
<td>111000</td>
<td>VX</td>
<td>166</td>
<td></td>
<td>frp[.a]</td>
<td>v2.02</td>
<td></td>
<td></td>
<td>Floating Round To Integer Zero</td>
</tr>
<tr>
<td>111000</td>
<td>VX</td>
<td>155</td>
<td></td>
<td>fsgntl[.a]</td>
<td>v2.02</td>
<td>Floating Reciprocal Square Root Estimate Single</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111000</td>
<td>XL</td>
<td>956</td>
<td></td>
<td>rfsid[.a]</td>
<td>v2.02</td>
<td>H V Return From interrupt Doubleword Hypervisor</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111000</td>
<td>XL</td>
<td>97</td>
<td></td>
<td>rfsnid[.a]</td>
<td>v2.02</td>
<td>Population Count Byte</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111000</td>
<td>FX</td>
<td>122</td>
<td></td>
<td>mfoct[.a]</td>
<td>v2.01</td>
<td>Move From One CR Field</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111000</td>
<td>FX</td>
<td>121</td>
<td></td>
<td>mfruitc[.a]</td>
<td>v2.01</td>
<td>Move To One CR Field</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111000</td>
<td>X</td>
<td>1031</td>
<td></td>
<td>sbmfe[.a]</td>
<td>v2.00</td>
<td>P SLB Move From Entry ESIID</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111000</td>
<td>X</td>
<td>1030</td>
<td></td>
<td>sbmfev[.a]</td>
<td>v2.00</td>
<td>P SLB Move From Entry VSID</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111000</td>
<td>X</td>
<td>1029</td>
<td></td>
<td>sbmfb[.a]</td>
<td>v2.00</td>
<td>P SLB Move To Entry</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111000</td>
<td>X</td>
<td>99</td>
<td></td>
<td>cndtz[.a]</td>
<td>PPC</td>
<td>SR Count Leading Zeros Doubleword</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111000</td>
<td>X</td>
<td>852</td>
<td></td>
<td>dcdf[.a]</td>
<td>PPC</td>
<td>Data Cache Block Flush</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111000</td>
<td>X</td>
<td>851</td>
<td></td>
<td>dcdb[.a]</td>
<td>PPC</td>
<td>Data Cache Block Store</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111000</td>
<td>X</td>
<td>849</td>
<td></td>
<td>dcbt[.a]</td>
<td>PPC</td>
<td>Data Cache Block Touch</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111000</td>
<td>X</td>
<td>850</td>
<td></td>
<td>dcblt[.a]</td>
<td>PPC</td>
<td>Data Cache Block Touch for Store</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111000</td>
<td>X</td>
<td>81</td>
<td></td>
<td>div[.a]</td>
<td>PPC</td>
<td>SR Divide Doubleword</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111000</td>
<td>X</td>
<td>81</td>
<td></td>
<td>divdp[.a]</td>
<td>PPC</td>
<td>SR Divide Doubleword Unsigned</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111000</td>
<td>X</td>
<td>74</td>
<td></td>
<td>divw[.a]</td>
<td>PPC</td>
<td>SR Divide Word</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111000</td>
<td>X</td>
<td>74</td>
<td></td>
<td>divup[.a]</td>
<td>PPC</td>
<td>SR Divide Word Unsigned</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111000</td>
<td>X</td>
<td>875</td>
<td></td>
<td>eis[.a]</td>
<td>PPC</td>
<td>Enforce In-order Execution of I/O</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111000</td>
<td>X</td>
<td>96</td>
<td></td>
<td>extsb[.a]</td>
<td>PPC</td>
<td>SR Extend Sign Byte</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111000</td>
<td>X</td>
<td>96</td>
<td></td>
<td>extsw[.a]</td>
<td>PPC</td>
<td>SR Extend Sign Word</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111000</td>
<td>A</td>
<td>152</td>
<td></td>
<td>fadd[.a]</td>
<td>PPC</td>
<td>Floating Add Single</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111000</td>
<td>X</td>
<td>163</td>
<td></td>
<td>fcll[.a]</td>
<td>PPC</td>
<td>Floating Convert with round Signed Doubleword to Double-Precision format</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 89. Power ISA AS Instruction Set Sorted by Version (Sheet 14 of 18)
Figure 89. Power ISA AS Instruction Set Sorted by Version (Sheet 15 of 18)
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>Book</th>
<th>Mnemonic</th>
<th>Version</th>
<th>Privilege</th>
<th>Mode Dep</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:5 6:10 13:15 16:21 25:28</td>
<td>X</td>
<td>I</td>
<td>161</td>
<td>fctw[.]</td>
<td>P2</td>
<td>Floating Convert with round Double-Precision To Signed Word format</td>
<td></td>
</tr>
<tr>
<td>0:5 6:10 13:15 16:21 25:28</td>
<td>X</td>
<td>I</td>
<td>162</td>
<td>fctwz[.]</td>
<td>P2</td>
<td>Floating Convert with round to Zero Double-Precision To Signed Word format</td>
<td></td>
</tr>
<tr>
<td>0:5 6:10 13:15 16:21 25:28</td>
<td>A</td>
<td>I</td>
<td>154</td>
<td>fsgt[.]</td>
<td>P2</td>
<td>Floating Square Root</td>
<td></td>
</tr>
<tr>
<td>0:5 6:10 13:15 16:21 25:28</td>
<td>XO</td>
<td>I</td>
<td>69</td>
<td>add[.][o]</td>
<td>P1</td>
<td>SR</td>
<td>Add</td>
</tr>
<tr>
<td>0:5 6:10 13:15 16:21 25:28</td>
<td>XO</td>
<td>I</td>
<td>70</td>
<td>addc[.][o]</td>
<td>P1</td>
<td>SR</td>
<td>Add Carrying</td>
</tr>
<tr>
<td>0:5 6:10 13:15 16:21 25:28</td>
<td>XO</td>
<td>I</td>
<td>71</td>
<td>addde[.][o]</td>
<td>P1</td>
<td>SR</td>
<td>Add Extended</td>
</tr>
<tr>
<td>0:5 6:10 13:15 16:21 25:28</td>
<td>D</td>
<td>I</td>
<td>67</td>
<td>add</td>
<td>P1</td>
<td>Add Immediate</td>
<td></td>
</tr>
<tr>
<td>0:5 6:10 13:15 16:21 25:28</td>
<td>D</td>
<td>I</td>
<td>69</td>
<td>addc</td>
<td>P1</td>
<td>SR</td>
<td>Add Immediate Carrying</td>
</tr>
<tr>
<td>0:5 6:10 13:15 16:21 25:28</td>
<td>D</td>
<td>I</td>
<td>67</td>
<td>addds</td>
<td>P1</td>
<td>Add Immediate Shifted</td>
<td></td>
</tr>
<tr>
<td>0:5 6:10 13:15 16:21 25:28</td>
<td>XO</td>
<td>I</td>
<td>71</td>
<td>addme[.][o]</td>
<td>P1</td>
<td>SR</td>
<td>Add to Minus One Extended</td>
</tr>
<tr>
<td>0:5 6:10 13:15 16:21 25:28</td>
<td>XO</td>
<td>I</td>
<td>72</td>
<td>addze[.][o]</td>
<td>P1</td>
<td>SR</td>
<td>Add to Zero Extended</td>
</tr>
<tr>
<td>0:5 6:10 13:15 16:21 25:28</td>
<td>X</td>
<td>I</td>
<td>94</td>
<td>and[.]</td>
<td>P1</td>
<td>SR</td>
<td>AND</td>
</tr>
<tr>
<td>0:5 6:10 13:15 16:21 25:28</td>
<td>X</td>
<td>I</td>
<td>88</td>
<td>andc[.][.]</td>
<td>P1</td>
<td>SR</td>
<td>AND with Complement</td>
</tr>
<tr>
<td>0:5 6:10 13:15 16:21 25:28</td>
<td>D</td>
<td>I</td>
<td>92</td>
<td>andis</td>
<td>P1</td>
<td>SR</td>
<td>AND Immediate &amp; record</td>
</tr>
<tr>
<td>0:5 6:10 13:15 16:21 25:28</td>
<td>D</td>
<td>I</td>
<td>92</td>
<td>andic</td>
<td>P1</td>
<td>SR</td>
<td>AND Immediate &amp; record</td>
</tr>
<tr>
<td>0:5 6:10 13:15 16:21 25:28</td>
<td>I</td>
<td>I</td>
<td>37</td>
<td>b[.][a]</td>
<td>P1</td>
<td>Branch &amp; [Link] [Absolute]</td>
<td></td>
</tr>
<tr>
<td>0:5 6:10 13:15 16:21 25:28</td>
<td>I</td>
<td>I</td>
<td>37</td>
<td>bcond[.]</td>
<td>P1</td>
<td>CT</td>
<td>Branch Conditional &amp; [Link] [Absolute]</td>
</tr>
<tr>
<td>0:5 6:10 13:15 16:21 25:28</td>
<td>XL</td>
<td>I</td>
<td>38</td>
<td>bcond[.]</td>
<td>P1</td>
<td>CT</td>
<td>Branch Conditional to CTR &amp; [Link]</td>
</tr>
<tr>
<td>0:5 6:10 13:15 16:21 25:28</td>
<td>XL</td>
<td>I</td>
<td>38</td>
<td>bcond[.]</td>
<td>P1</td>
<td>CT</td>
<td>Branch Conditional to LR &amp; [Link]</td>
</tr>
<tr>
<td>0:5 6:10 13:15 16:21 25:28</td>
<td>X</td>
<td>I</td>
<td>85</td>
<td>cmp</td>
<td>P1</td>
<td>Compare</td>
<td></td>
</tr>
<tr>
<td>0:5 6:10 13:15 16:21 25:28</td>
<td>X</td>
<td>I</td>
<td>85</td>
<td>cmpi</td>
<td>P1</td>
<td>Compare Immediate</td>
<td></td>
</tr>
<tr>
<td>0:5 6:10 13:15 16:21 25:28</td>
<td>X</td>
<td>I</td>
<td>86</td>
<td>cmpl</td>
<td>P1</td>
<td>Compare Logical</td>
<td></td>
</tr>
<tr>
<td>0:5 6:10 13:15 16:21 25:28</td>
<td>D</td>
<td>I</td>
<td>86</td>
<td>cmpl</td>
<td>P1</td>
<td>Compare Logical Immediate</td>
<td></td>
</tr>
<tr>
<td>0:5 6:10 13:15 16:21 25:28</td>
<td>X</td>
<td>I</td>
<td>96</td>
<td>cntlzw[.]</td>
<td>P1</td>
<td>SR</td>
<td>Count Leading Zeros Word</td>
</tr>
<tr>
<td>0:5 6:10 13:15 16:21 25:28</td>
<td>XL</td>
<td>I</td>
<td>40</td>
<td>crand</td>
<td>P1</td>
<td>CR AND</td>
<td></td>
</tr>
<tr>
<td>0:5 6:10 13:15 16:21 25:28</td>
<td>XL</td>
<td>I</td>
<td>41</td>
<td>crandc</td>
<td>P1</td>
<td>CR AND with Complement</td>
<td></td>
</tr>
<tr>
<td>0:5 6:10 13:15 16:21 25:28</td>
<td>XL</td>
<td>I</td>
<td>42</td>
<td>crenq</td>
<td>P1</td>
<td>CR Equivalent</td>
<td></td>
</tr>
<tr>
<td>0:5 6:10 13:15 16:21 25:28</td>
<td>XL</td>
<td>I</td>
<td>43</td>
<td>crand</td>
<td>P1</td>
<td>CR NAND</td>
<td></td>
</tr>
<tr>
<td>0:5 6:10 13:15 16:21 25:28</td>
<td>XL</td>
<td>I</td>
<td>41</td>
<td>cmor</td>
<td>P1</td>
<td>CR OR</td>
<td></td>
</tr>
<tr>
<td>0:5 6:10 13:15 16:21 25:28</td>
<td>XL</td>
<td>I</td>
<td>41</td>
<td>cror</td>
<td>P1</td>
<td>CR OR with Complement</td>
<td></td>
</tr>
<tr>
<td>0:5 6:10 13:15 16:21 25:28</td>
<td>XL</td>
<td>I</td>
<td>40</td>
<td>cxor</td>
<td>P1</td>
<td>CR XOR</td>
<td></td>
</tr>
<tr>
<td>0:5 6:10 13:15 16:21 25:28</td>
<td>X</td>
<td>I</td>
<td>851</td>
<td>dbcz</td>
<td>P1</td>
<td>SR</td>
<td>Data Cache Block Zero</td>
</tr>
<tr>
<td>0:5 6:10 13:15 16:21 25:28</td>
<td>X</td>
<td>I</td>
<td>95</td>
<td>eqv[.]</td>
<td>P1</td>
<td>SR</td>
<td>Equivalent</td>
</tr>
<tr>
<td>0:5 6:10 13:15 16:21 25:28</td>
<td>X</td>
<td>I</td>
<td>96</td>
<td>extsh[.]</td>
<td>P1</td>
<td>SR</td>
<td>Extend Sign Halfword</td>
</tr>
<tr>
<td>0:5 6:10 13:15 16:21 25:28</td>
<td>X</td>
<td>I</td>
<td>150</td>
<td>fabs[.]</td>
<td>P1</td>
<td>Floating Absolute</td>
<td></td>
</tr>
<tr>
<td>0:5 6:10 13:15 16:21 25:28</td>
<td>A</td>
<td>I</td>
<td>152</td>
<td>fadd[.]</td>
<td>P1</td>
<td>Floating Add</td>
<td></td>
</tr>
<tr>
<td>0:5 6:10 13:15 16:21 25:28</td>
<td>A</td>
<td>I</td>
<td>167</td>
<td>fcmpo</td>
<td>P1</td>
<td>Floating Compare Ordered</td>
<td></td>
</tr>
<tr>
<td>0:5 6:10 13:15 16:21 25:28</td>
<td>I</td>
<td>I</td>
<td>167</td>
<td>fcmpu</td>
<td>P1</td>
<td>Floating Compare Unordered</td>
<td></td>
</tr>
<tr>
<td>0:5 6:10 13:15 16:21 25:28</td>
<td>X</td>
<td>I</td>
<td>150</td>
<td>fmr[.]</td>
<td>P1</td>
<td>Floating Move Register</td>
<td></td>
</tr>
<tr>
<td>0:5 6:10 13:15 16:21 25:28</td>
<td>A</td>
<td>I</td>
<td>158</td>
<td>fmsub[.]</td>
<td>P1</td>
<td>Floating Multiply-Subtract</td>
<td></td>
</tr>
<tr>
<td>0:5 6:10 13:15 16:21 25:28</td>
<td>X</td>
<td>I</td>
<td>150</td>
<td>fnabs[.]</td>
<td>P1</td>
<td>Floating Negative Absolute Value</td>
<td></td>
</tr>
<tr>
<td>0:5 6:10 13:15 16:21 25:28</td>
<td>X</td>
<td>I</td>
<td>150</td>
<td>fneg[.]</td>
<td>P1</td>
<td>Floating Negate</td>
<td></td>
</tr>
<tr>
<td>0:5 6:10 13:15 16:21 25:28</td>
<td>A</td>
<td>I</td>
<td>158</td>
<td>fnmad[.]</td>
<td>P1</td>
<td>Floating Negative Multiply-Add</td>
<td></td>
</tr>
<tr>
<td>0:5 6:10 13:15 16:21 25:28</td>
<td>A</td>
<td>I</td>
<td>158</td>
<td>fnmsub[.]</td>
<td>P1</td>
<td>Floating Negative Multiply-Subtract</td>
<td></td>
</tr>
<tr>
<td>0:5 6:10 13:15 16:21 25:28</td>
<td>A</td>
<td>I</td>
<td>159</td>
<td>frsp[.]</td>
<td>P1</td>
<td>Floating Round to Single-Precision</td>
<td></td>
</tr>
<tr>
<td>0:5 6:10 13:15 16:21 25:28</td>
<td>A</td>
<td>I</td>
<td>152</td>
<td>fsub[.]</td>
<td>P1</td>
<td>Floating Subtract</td>
<td></td>
</tr>
<tr>
<td>0:5 6:10 13:15 16:21 25:28</td>
<td>XL</td>
<td>I</td>
<td>863</td>
<td>sync</td>
<td>P1</td>
<td>Instruction Synchronize</td>
<td></td>
</tr>
<tr>
<td>0:5 6:10 13:15 16:21 25:28</td>
<td>D</td>
<td>I</td>
<td>48</td>
<td>lbz</td>
<td>P1</td>
<td>Load Byte &amp; Zero</td>
<td></td>
</tr>
<tr>
<td>0:5 6:10 13:15 16:21 25:28</td>
<td>D</td>
<td>I</td>
<td>48</td>
<td>lbzu</td>
<td>P1</td>
<td>Load Byte &amp; Zero with Update</td>
<td></td>
</tr>
<tr>
<td>0:5 6:10 13:15 16:21 25:28</td>
<td>X</td>
<td>I</td>
<td>48</td>
<td>lbzux</td>
<td>P1</td>
<td>Load Byte &amp; Zero with Update Indexed</td>
<td></td>
</tr>
</tbody>
</table>

Figure 89. Power ISA AS Instruction Set Sorted by Version (Sheet 16 of 18)
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>Book</th>
<th>Page</th>
<th>Mnemonic</th>
<th>Version</th>
<th>Privilege</th>
<th>Mode Dep</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X I</td>
<td>48</td>
<td>fbx</td>
<td>P1</td>
<td>Load Byte &amp; Zero Indexed</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>D I</td>
<td>142</td>
<td>lfd</td>
<td>P1</td>
<td>Load Floating Double</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>D I</td>
<td>142</td>
<td>lfd</td>
<td>P1</td>
<td>Load Floating Double with Update</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X I</td>
<td>143</td>
<td>lbux</td>
<td>P1</td>
<td>Load Floating Double with Update Indexed</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X I</td>
<td>142</td>
<td>lbux</td>
<td>P1</td>
<td>Load Floating Double Indexed</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>D I</td>
<td>140</td>
<td>lfs</td>
<td>P1</td>
<td>Load Floating Single</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>D I</td>
<td>141</td>
<td>lfsu</td>
<td>P1</td>
<td>Load Floating Single with Update</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X I</td>
<td>142</td>
<td>lfsu</td>
<td>P1</td>
<td>Load Floating Single with Update Indexed</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X I</td>
<td>141</td>
<td>lfs</td>
<td>P1</td>
<td>Load Floating Single Indexed</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>D I</td>
<td>50</td>
<td>lha</td>
<td>P1</td>
<td>Load Halfword Algebraic</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>D I</td>
<td>50</td>
<td>lha</td>
<td>P1</td>
<td>Load Halfword Algebraic with Update</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X I</td>
<td>50</td>
<td>lhau</td>
<td>P1</td>
<td>Load Halfword Algebraic with Update Indexed</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X I</td>
<td>50</td>
<td>lhax</td>
<td>P1</td>
<td>Load Halfword Algebraic Indexed</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X I</td>
<td>60</td>
<td>hbux</td>
<td>P1</td>
<td>Load Halfword Byte-Reverse Indexed</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>D I</td>
<td>49</td>
<td>lhz</td>
<td>P1</td>
<td>Load Halfword &amp; Zero</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>D I</td>
<td>49</td>
<td>lhz</td>
<td>P1</td>
<td>Load Halfword &amp; Zero with Update</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X I</td>
<td>49</td>
<td>lhux</td>
<td>P1</td>
<td>Load Halfword &amp; Zero with Update Indexed</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X I</td>
<td>49</td>
<td>lhux</td>
<td>P1</td>
<td>Load Halfword &amp; Zero Indexed</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>D I</td>
<td>62</td>
<td>lwz</td>
<td>P1</td>
<td>Load Multiple Word</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X I</td>
<td>64</td>
<td>ltwi</td>
<td>P1</td>
<td>Load String Word Immediate</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X I</td>
<td>64</td>
<td>ltwz</td>
<td>P1</td>
<td>Load String Word Indexed</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X I</td>
<td>60</td>
<td>lwr</td>
<td>P1</td>
<td>Load Word Byte-Reverse Indexed</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>D I</td>
<td>51</td>
<td>lwz</td>
<td>P1</td>
<td>Load Word &amp; Zero</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>D I</td>
<td>51</td>
<td>lwz</td>
<td>P1</td>
<td>Load Word &amp; Zero with Update</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X I</td>
<td>51</td>
<td>lwzxx</td>
<td>P1</td>
<td>Load Word &amp; Zero with Update Indexed</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X I</td>
<td>51</td>
<td>lwzxx</td>
<td>P1</td>
<td>Load Word &amp; Zero Indexed</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X I</td>
<td>51</td>
<td>lwzxx</td>
<td>P1</td>
<td>Load Word &amp; Zero Indexed</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X I</td>
<td>41</td>
<td>mcrf</td>
<td>P1</td>
<td>Move CR Field</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X I</td>
<td>171</td>
<td>mcrfs</td>
<td>P1</td>
<td>Move To CR from FPSCR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X I</td>
<td>170</td>
<td>mcrfs</td>
<td>P1</td>
<td>Move From CR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X I</td>
<td>170</td>
<td>mcrfs</td>
<td>P1</td>
<td>Move From FPSCR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X I</td>
<td>119</td>
<td>mtfsr</td>
<td>P1</td>
<td>Move From MSR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X I</td>
<td>975</td>
<td>mtfsr</td>
<td>P1</td>
<td>Move From MSR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X I</td>
<td>975</td>
<td>mtfsr</td>
<td>P1</td>
<td>Move From SPR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X I</td>
<td>975</td>
<td>mtfsr</td>
<td>P1</td>
<td>Move From SPR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X I</td>
<td>121</td>
<td>mtcrf</td>
<td>P1</td>
<td>Move To CR Fields</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X I</td>
<td>173</td>
<td>mtfsrb[ ]</td>
<td>P1</td>
<td>Move To FPSCR Bit 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X I</td>
<td>173</td>
<td>mtfsrb[ ]</td>
<td>P1</td>
<td>Move To FPSCR Bit 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X I</td>
<td>121</td>
<td>mtsr</td>
<td>P1</td>
<td>Move To FPSCR Fields</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X I</td>
<td>172</td>
<td>mtfsrb[ ]</td>
<td>P1</td>
<td>Move To FPSCR Field Immediate</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X I</td>
<td>172</td>
<td>mtfsrb[ ]</td>
<td>P1</td>
<td>Move To FPSCR Field Immediate</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X I</td>
<td>117</td>
<td>mtfsr</td>
<td>P1</td>
<td>Move To MSR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X I</td>
<td>974</td>
<td>mtfsr</td>
<td>P1</td>
<td>Move To SPR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X I</td>
<td>73</td>
<td>multi</td>
<td>P1</td>
<td>Multiply Low Immediate</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X I</td>
<td>94</td>
<td>nanol[]</td>
<td>P1</td>
<td>SR NAND</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X I</td>
<td>72</td>
<td>nego[]</td>
<td>P1</td>
<td>SR Negate</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X I</td>
<td>95</td>
<td>nor[]</td>
<td>P1</td>
<td>SR NOR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X I</td>
<td>94</td>
<td>or[]</td>
<td>P1</td>
<td>SR OR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X I</td>
<td>95</td>
<td>orc[]</td>
<td>P1</td>
<td>SR OR with Complement</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>D I</td>
<td>92</td>
<td>or[]</td>
<td>P1</td>
<td>OR Immediate</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>D I</td>
<td>93</td>
<td>or[]</td>
<td>P1</td>
<td>OR Immediate Shifted</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>M I</td>
<td>103</td>
<td>hvwm[]</td>
<td>P1</td>
<td>SR Rotate Left Word Immediate then Mask Insert</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>M I</td>
<td>102</td>
<td>hvwm[]</td>
<td>P1</td>
<td>SR Rotate Left Word Immediate then AND with Mask</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>M I</td>
<td>103</td>
<td>hvwm[]</td>
<td>P1</td>
<td>SR Rotate Left Word then AND with Mask</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X I</td>
<td>107</td>
<td>sa[ ]</td>
<td>P1</td>
<td>SR Shift Left Word</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X I</td>
<td>106</td>
<td>sa[ ]</td>
<td>P1</td>
<td>SR Shift Right Algebraic Word</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X I</td>
<td>106</td>
<td>sa[ ]</td>
<td>P1</td>
<td>SR Shift Right Algebraic Word Immediate</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X I</td>
<td>107</td>
<td>swr[ ]</td>
<td>P1</td>
<td>SR Shift Right Word</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 89. Power ISA AS Instruction Set Sorted by Version (Sheet 17 of 18)
Figure 89. Power ISA AS Instruction Set Sorted by Version (Sheet 18 of 18)

1. Key to Instruction column.
   - 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 having a value 1.

2. Key to Version column.
   - P1 Instruction introduced in the POWER Architecture.
   - P2 Instruction introduced in the POWER2 Architecture.
   - PPC Instruction introduced in the PowerPC Architecture prior to v2.00.
   - v2.00 Instruction introduced in the PowerPC Architecture Version 2.00.
   - v2.01 Instruction introduced in the PowerPC Architecture Version 2.01.
   - v2.02 Instruction introduced in the PowerPC Architecture Version 2.02.
   - v2.03 Instruction introduced in the Power ISA Architecture Version 2.03.
   - v2.04 Instruction introduced in the Power ISA Architecture Version 2.04.
   - v2.05 Instruction introduced in the Power ISA Architecture Version 2.05.
   - v2.06 Instruction introduced in the Power ISA Architecture Version 2.06.
   - v2.07 Instruction introduced in the Power ISA Architecture Version 2.07.
   - v3.0 Instruction introduced in the Power ISA Architecture Version 3.0.
   - v3.0B Instruction introduced in the Power ISA Architecture Version 3.0B.
3. Key to Privilege column.
   - **P** Denotes an instruction that is treated as privileged.
   - **O** Denotes an instruction that is treated as privileged or nonprivileged (or hypervisor, for mtspr), depending on the SPR or PMR number.
   - **PI** Denotes an instruction that is illegal in privileged state.
   - **H** Denotes an instruction that can be executed only in hypervisor state.
   - **U** Denotes an instruction that can be executed only in ultravisor state.

4. Key to Mode Dependency column.
   Except as described below and in Section 1.11.3, “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.
   - **32** The instruction can be executed only in 32-bit mode.
   - **64** The instruction can be executed only in 64-bit mode.
# Appendix F. 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>Format</th>
<th>Book</th>
<th>Page</th>
<th>Mnemonic</th>
<th>Version</th>
<th>Privilege</th>
<th>Mode Dep</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:5</td>
<td>6:10</td>
<td>11:15</td>
<td>16:20</td>
<td>21:25</td>
<td>26:31</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>69</td>
<td>add[o]</td>
<td>P1</td>
<td>SR</td>
<td>Add</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>70</td>
<td>addc[o]</td>
<td>P1</td>
<td>SR</td>
<td>Add Carrying</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>71</td>
<td>adde[o]</td>
<td>P1</td>
<td>SR</td>
<td>Add Extended</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>72</td>
<td>addg6s</td>
<td>v3.0B</td>
<td>Add Extended using alternate carry</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>69</td>
<td>addi</td>
<td>P1</td>
<td></td>
<td>Add Immediate</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>69</td>
<td>addic</td>
<td>P1</td>
<td>SR</td>
<td>Add Immediate Carrying</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>69</td>
<td>addic.</td>
<td>P1</td>
<td>SR</td>
<td>Add Immediate Carrying &amp; record</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>67</td>
<td>add</td>
<td>P1</td>
<td></td>
<td>Add Immediate Shifted</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>71</td>
<td>addme[o]</td>
<td>P1</td>
<td></td>
<td>Add to Minus One Extended</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>68</td>
<td>addpcis</td>
<td>v3.0</td>
<td>Add PC Immediate Shifted</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>69</td>
<td>and</td>
<td>P1</td>
<td>SR</td>
<td>AND</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>72</td>
<td>andc</td>
<td>P1</td>
<td>SR</td>
<td>AND with Complement</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>71</td>
<td>andi</td>
<td>P1</td>
<td>SR</td>
<td>AND Immediate &amp; record</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>72</td>
<td>andis</td>
<td>P1</td>
<td>SR</td>
<td>AND Immediate Shifted &amp; record</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>37</td>
<td>b[l]a</td>
<td>P1</td>
<td>CT</td>
<td>Branch (Link) [Absolute]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>37</td>
<td>bc[l]a</td>
<td>P1</td>
<td>CT</td>
<td>Branch Conditional (Link) [Absolute]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>38</td>
<td>bctar</td>
<td>v3.0</td>
<td>Branch Conditional to BTAR [Link]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>38</td>
<td>bcct[ei]</td>
<td>P1</td>
<td>CT</td>
<td>Branch Conditional to CTR (Link)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>348</td>
<td>bcdaddc</td>
<td>v2.07</td>
<td>Decimal Add Modulo &amp; record</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>350</td>
<td>bcdchn</td>
<td>v3.0</td>
<td>Decimal Convert From National &amp; record</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>354</td>
<td>bcdcfsq</td>
<td>v3.0</td>
<td>Decimal Convert From Signed Quadword &amp; record</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>351</td>
<td>bcdcz</td>
<td>v3.0</td>
<td>Decimal Convert From Zoned &amp; record</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>356</td>
<td>bcdcp</td>
<td>v3.0</td>
<td>Decimal CopySign &amp; record</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>352</td>
<td>bcdclh</td>
<td>v3.0</td>
<td>Decimal Convert To National &amp; record</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>354</td>
<td>bcdcq</td>
<td>v3.0</td>
<td>Decimal Convert To Signed Quadword &amp; record</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>353</td>
<td>bcdct</td>
<td>v3.0</td>
<td>Decimal Convert To Zoned &amp; record</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>357</td>
<td>bcds</td>
<td>v3.0</td>
<td>Decimal Shift &amp; record</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>356</td>
<td>bcdsetsign</td>
<td>v3.0</td>
<td>Decimal Set Sign &amp; record</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>359</td>
<td>bcdei</td>
<td>v3.0</td>
<td>Decimal Shift &amp; Round &amp; record</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>346</td>
<td>bcdsub</td>
<td>v2.07</td>
<td>Decimal Subtract Module &amp; record</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>360</td>
<td>bcdtrunc</td>
<td>v3.0</td>
<td>Decimal Truncate &amp; record</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>358</td>
<td>bcderus</td>
<td>v3.0</td>
<td>Decimal Unsigned Shift &amp; record</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>361</td>
<td>bcdtrunc</td>
<td>v3.0</td>
<td>Decimal Uns unsigned Truncate &amp; record</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>359</td>
<td>bcdsetsign</td>
<td>v3.0</td>
<td>Decimal Set Sign &amp; record</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>359</td>
<td>bcdei</td>
<td>v3.0</td>
<td>Decimal Shift &amp; Round &amp; record</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>346</td>
<td>bcdsub</td>
<td>v2.07</td>
<td>Decimal Subtract Module &amp; record</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>360</td>
<td>bcdtrunc</td>
<td>v3.0</td>
<td>Decimal Truncate &amp; record</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>358</td>
<td>bcderus</td>
<td>v3.0</td>
<td>Decimal Unsigned Shift &amp; record</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>361</td>
<td>bcdtrunc</td>
<td>v3.0</td>
<td>Decimal Uns unsigned Truncate &amp; record</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>359</td>
<td>bcdsetsign</td>
<td>v3.0</td>
<td>Decimal Set Sign &amp; record</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>359</td>
<td>bcdei</td>
<td>v3.0</td>
<td>Decimal Shift &amp; Round &amp; record</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>346</td>
<td>bcdsub</td>
<td>v2.07</td>
<td>Decimal Subtract Module &amp; record</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>360</td>
<td>bcdtrunc</td>
<td>v3.0</td>
<td>Decimal Truncate &amp; record</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>358</td>
<td>bcderus</td>
<td>v3.0</td>
<td>Decimal Unsigned Shift &amp; record</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>361</td>
<td>bcdtrunc</td>
<td>v3.0</td>
<td>Decimal Uns unsigned Truncate &amp; record</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>359</td>
<td>bcdsetsign</td>
<td>v3.0</td>
<td>Decimal Set Sign &amp; record</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>359</td>
<td>bcdei</td>
<td>v3.0</td>
<td>Decimal Shift &amp; Round &amp; record</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>346</td>
<td>bcdsub</td>
<td>v2.07</td>
<td>Decimal Subtract Module &amp; record</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>360</td>
<td>bcdtrunc</td>
<td>v3.0</td>
<td>Decimal Truncate &amp; record</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>358</td>
<td>bcderus</td>
<td>v3.0</td>
<td>Decimal Uns signed Truncate &amp; record</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>361</td>
<td>bcdtrunc</td>
<td>v3.0</td>
<td>Decimal Unsigne Truncate &amp; record</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>359</td>
<td>bcdsetsign</td>
<td>v3.0</td>
<td>Decimal Set Sign &amp; record</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>359</td>
<td>bcdei</td>
<td>v3.0</td>
<td>Decimal Shift &amp; Round &amp; record</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>346</td>
<td>bcdsub</td>
<td>v2.07</td>
<td>Decimal Subtract Module &amp; record</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>360</td>
<td>bcdtrunc</td>
<td>v3.0</td>
<td>Decimal Truncate &amp; record</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>358</td>
<td>bcderus</td>
<td>v3.0</td>
<td>Decimal Uns signed Truncate &amp; record</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>361</td>
<td>bcdtrunc</td>
<td>v3.0</td>
<td>Decimal Unsigne Truncate &amp; record</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 90. Power ISA AS Instruction Set Sorted by Mnemonic (Sheet 1 of 18)
<table>
<thead>
<tr>
<th>Instruction1</th>
<th>Format</th>
<th>Book</th>
<th>Page</th>
<th>Mnemonic</th>
<th>Version2</th>
<th>Privilege2</th>
<th>Mode Dep4</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0:5</td>
<td>8:10</td>
<td>11:15</td>
<td>16:20</td>
<td>21:25</td>
<td>26:33</td>
<td></td>
<td></td>
</tr>
<tr>
<td>011111</td>
<td>011111</td>
<td>X</td>
<td>1</td>
<td>cmpb</td>
<td>v2.05</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011111</td>
<td>001111</td>
<td>X</td>
<td>1</td>
<td>cmpseqb</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000111</td>
<td>D</td>
<td>85</td>
<td>cmpi</td>
<td>P1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000111</td>
<td>D</td>
<td>86</td>
<td>cmpi</td>
<td>P1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001111</td>
<td>011111</td>
<td>D</td>
<td>87</td>
<td>cmbpb</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011111</td>
<td>011111</td>
<td>X</td>
<td>99</td>
<td>critizw[]</td>
<td>PPC</td>
<td>SR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>011111</td>
<td>011111</td>
<td>X</td>
<td>98</td>
<td>critizw[]</td>
<td>PPC</td>
<td>SR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>011111</td>
<td>011111</td>
<td>X</td>
<td>99</td>
<td>critizd[]</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011111</td>
<td>011111</td>
<td>X</td>
<td>99</td>
<td>critizd[]</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011111</td>
<td>1000110</td>
<td>X</td>
<td>855</td>
<td>copy</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011111</td>
<td>0010010</td>
<td>X</td>
<td>40</td>
<td>crand</td>
<td>P1</td>
<td></td>
<td>CR AND</td>
<td></td>
</tr>
<tr>
<td>011111</td>
<td>0101010</td>
<td>X</td>
<td>41</td>
<td>crandc</td>
<td>P1</td>
<td></td>
<td>CR AND with Complement</td>
<td></td>
</tr>
<tr>
<td>011111</td>
<td>0101010</td>
<td>X</td>
<td>41</td>
<td>creqv</td>
<td>P1</td>
<td></td>
<td>CR Equivalent</td>
<td></td>
</tr>
<tr>
<td>011111</td>
<td>0011110</td>
<td>X</td>
<td>40</td>
<td>cmst</td>
<td>P1</td>
<td></td>
<td>CR NAND</td>
<td></td>
</tr>
<tr>
<td>011111</td>
<td>0001110</td>
<td>X</td>
<td>41</td>
<td>cmor</td>
<td>P1</td>
<td></td>
<td>CR NOR</td>
<td></td>
</tr>
<tr>
<td>011111</td>
<td>0110010</td>
<td>X</td>
<td>40</td>
<td>cxor</td>
<td>P1</td>
<td></td>
<td>CR OR</td>
<td></td>
</tr>
<tr>
<td>011111</td>
<td>0110010</td>
<td>X</td>
<td>41</td>
<td>crorc</td>
<td>P1</td>
<td></td>
<td>CR OR with Complement</td>
<td></td>
</tr>
<tr>
<td>011111</td>
<td>0110010</td>
<td>X</td>
<td>40</td>
<td>cnor</td>
<td>P1</td>
<td></td>
<td>CR XOR</td>
<td></td>
</tr>
<tr>
<td>011111</td>
<td>0000010</td>
<td>X</td>
<td>193</td>
<td>dadd[]</td>
<td>v2.05</td>
<td>DFP Add</td>
<td></td>
<td></td>
</tr>
<tr>
<td>011111</td>
<td>0000010</td>
<td>X</td>
<td>193</td>
<td>dadd[]</td>
<td>v2.05</td>
<td>DFP Add Quad</td>
<td></td>
<td></td>
</tr>
<tr>
<td>011111</td>
<td>0000110</td>
<td>X</td>
<td>178</td>
<td>dcmpb</td>
<td>v4.0</td>
<td>Deliver A Random Number</td>
<td></td>
<td></td>
</tr>
<tr>
<td>011111</td>
<td>0000110</td>
<td>X</td>
<td>852</td>
<td>dcmpb</td>
<td>PPC</td>
<td>Data Cache Block Flush</td>
<td></td>
<td></td>
</tr>
<tr>
<td>011111</td>
<td>0000110</td>
<td>X</td>
<td>851</td>
<td>dcmpb</td>
<td>PPC</td>
<td>Data Cache Block Store</td>
<td></td>
<td></td>
</tr>
<tr>
<td>011111</td>
<td>0000110</td>
<td>X</td>
<td>849</td>
<td>dcmpb</td>
<td>PPC</td>
<td>Data Cache Block Touch</td>
<td></td>
<td></td>
</tr>
<tr>
<td>011111</td>
<td>0000110</td>
<td>X</td>
<td>850</td>
<td>dcmpb</td>
<td>PPC</td>
<td>Data Cache Block Touch for Store</td>
<td></td>
<td></td>
</tr>
<tr>
<td>011111</td>
<td>0000110</td>
<td>X</td>
<td>851</td>
<td>dcmpb</td>
<td>P1</td>
<td>Data Cache Block Zero</td>
<td></td>
<td></td>
</tr>
<tr>
<td>011111</td>
<td>0000110</td>
<td>X</td>
<td>215</td>
<td>dcmpb</td>
<td>v2.06</td>
<td>DFP Convert From Fixed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>011111</td>
<td>0000110</td>
<td>X</td>
<td>215</td>
<td>dcmpb</td>
<td>v2.05</td>
<td>DFP Convert From Fixed Quad</td>
<td></td>
<td></td>
</tr>
<tr>
<td>011111</td>
<td>0010010</td>
<td>X</td>
<td>199</td>
<td>dcmpo</td>
<td>v2.05</td>
<td>DFP Compare Ordered</td>
<td></td>
<td></td>
</tr>
<tr>
<td>011111</td>
<td>0010010</td>
<td>X</td>
<td>199</td>
<td>dcmpo</td>
<td>v2.05</td>
<td>DFP Compare Ordered Quad</td>
<td></td>
<td></td>
</tr>
<tr>
<td>011111</td>
<td>0101010</td>
<td>X</td>
<td>198</td>
<td>dcmpo</td>
<td>v2.05</td>
<td>DFP Compare Unordered</td>
<td></td>
<td></td>
</tr>
<tr>
<td>011111</td>
<td>0101010</td>
<td>X</td>
<td>198</td>
<td>dcmpo</td>
<td>v2.05</td>
<td>DFP Compare Unordered Quad</td>
<td></td>
<td></td>
</tr>
<tr>
<td>011111</td>
<td>0101010</td>
<td>X</td>
<td>213</td>
<td>dcmpo</td>
<td>v2.05</td>
<td>DFP Convert To DFP Long</td>
<td></td>
<td></td>
</tr>
<tr>
<td>011111</td>
<td>0101010</td>
<td>X</td>
<td>213</td>
<td>dcmpo</td>
<td>v2.05</td>
<td>DFP Convert To Fixed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>011111</td>
<td>0101010</td>
<td>X</td>
<td>213</td>
<td>dcmpo</td>
<td>v2.05</td>
<td>DFP Convert To Fixed Quad</td>
<td></td>
<td></td>
</tr>
<tr>
<td>011111</td>
<td>0101010</td>
<td>X</td>
<td>213</td>
<td>dcmpo</td>
<td>v2.05</td>
<td>DFP Convert To DFP Extended</td>
<td></td>
<td></td>
</tr>
<tr>
<td>011111</td>
<td>0101010</td>
<td>X</td>
<td>217</td>
<td>dcmpo</td>
<td>v2.05</td>
<td>DFP Decode DPD To BCD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>011111</td>
<td>0101010</td>
<td>X</td>
<td>217</td>
<td>dcmpo</td>
<td>v2.05</td>
<td>DFP Decode DPD To BCD Quad</td>
<td></td>
<td></td>
</tr>
<tr>
<td>011111</td>
<td>0101010</td>
<td>X</td>
<td>196</td>
<td>dcmpo</td>
<td>v2.05</td>
<td>DFP Divide</td>
<td></td>
<td></td>
</tr>
<tr>
<td>011111</td>
<td>0101010</td>
<td>X</td>
<td>196</td>
<td>dcmpo</td>
<td>v2.05</td>
<td>DFP Divide Quad</td>
<td></td>
<td></td>
</tr>
<tr>
<td>011111</td>
<td>0101010</td>
<td>X</td>
<td>217</td>
<td>dcmpo</td>
<td>v2.05</td>
<td>DFP Encode BCD To DPD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>011111</td>
<td>0101010</td>
<td>X</td>
<td>217</td>
<td>dcmpo</td>
<td>v2.05</td>
<td>DFP Encode BCD To DPD Quad</td>
<td></td>
<td></td>
</tr>
<tr>
<td>011111</td>
<td>0101010</td>
<td>X</td>
<td>218</td>
<td>dcmpo</td>
<td>v2.05</td>
<td>DFP Insert Exponent</td>
<td></td>
<td></td>
</tr>
<tr>
<td>011111</td>
<td>0101010</td>
<td>X</td>
<td>218</td>
<td>dcmpo</td>
<td>v2.05</td>
<td>DFP Insert Exponent Quad</td>
<td></td>
<td></td>
</tr>
<tr>
<td>011111</td>
<td>0110010</td>
<td>X</td>
<td>81</td>
<td>divo[]</td>
<td>PPC</td>
<td>SR</td>
<td>Divide Doubleword</td>
<td></td>
</tr>
<tr>
<td>011111</td>
<td>0110010</td>
<td>X</td>
<td>82</td>
<td>divo[]</td>
<td>PPC</td>
<td>SR</td>
<td>Divide Doubleword Extended</td>
<td></td>
</tr>
<tr>
<td>011111</td>
<td>0110010</td>
<td>X</td>
<td>82</td>
<td>divo[]</td>
<td>PPC</td>
<td>SR</td>
<td>Divide Doubleword Extended Unsigned</td>
<td></td>
</tr>
<tr>
<td>011111</td>
<td>0110010</td>
<td>X</td>
<td>81</td>
<td>divo[]</td>
<td>PPC</td>
<td>SR</td>
<td>Divide Doubleword Unsigned</td>
<td></td>
</tr>
<tr>
<td>011111</td>
<td>0110010</td>
<td>X</td>
<td>74</td>
<td>divo[]</td>
<td>PPC</td>
<td>SR</td>
<td>Divide Word</td>
<td></td>
</tr>
<tr>
<td>011111</td>
<td>0110010</td>
<td>X</td>
<td>75</td>
<td>divo[]</td>
<td>PPC</td>
<td>SR</td>
<td>Divide Word Extended</td>
<td></td>
</tr>
<tr>
<td>011111</td>
<td>0110010</td>
<td>X</td>
<td>75</td>
<td>divo[]</td>
<td>PPC</td>
<td>SR</td>
<td>Divide Word Extended Unsigned</td>
<td></td>
</tr>
<tr>
<td>011111</td>
<td>0110010</td>
<td>X</td>
<td>74</td>
<td>divo[]</td>
<td>PPC</td>
<td>SR</td>
<td>Divide Word Unsigned</td>
<td></td>
</tr>
<tr>
<td>011111</td>
<td>0001110</td>
<td>X</td>
<td>195</td>
<td>simul[]</td>
<td>v2.05</td>
<td>DFP Multiply</td>
<td></td>
<td></td>
</tr>
<tr>
<td>011111</td>
<td>0001110</td>
<td>X</td>
<td>195</td>
<td>simul[]</td>
<td>v2.05</td>
<td>DFP Multiply Quad</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 90. Power ISA AS Instruction Set Sorted by Mnemonic (Sheet 2 of 18)
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>Book</th>
<th>Page</th>
<th>Mnemonic</th>
<th>Version</th>
<th>Privilege</th>
<th>Mode Dep</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>111011 ...... 000 00001</td>
<td>Z23</td>
<td>1</td>
<td>204</td>
<td>dqda[]</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>DFP Quantize</td>
</tr>
<tr>
<td>111011 ...... 000 00001</td>
<td>Z23</td>
<td>1</td>
<td>204</td>
<td>dqad[]</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>DFP Quantize Immediate</td>
</tr>
<tr>
<td>111011 ...... 000 00001</td>
<td>Z23</td>
<td>1</td>
<td>204</td>
<td>dqadq</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>DFP Quantize Immediate Quad</td>
</tr>
<tr>
<td>111011 ...... 000 00001</td>
<td>Z23</td>
<td>1</td>
<td>204</td>
<td>dquad[]</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>DFP Quantize Quad</td>
</tr>
<tr>
<td>111011 ...... 1110 00001</td>
<td>X</td>
<td>1</td>
<td>214</td>
<td>dsdp[]</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>DFP Round To DFP Long</td>
</tr>
<tr>
<td>111011 ...... 1110 00001</td>
<td>Z23</td>
<td>1</td>
<td>211</td>
<td>ddrin[]</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>DFP Round To FP Integer Without Inexact</td>
</tr>
<tr>
<td>111011 ...... 1110 00001</td>
<td>Z23</td>
<td>1</td>
<td>211</td>
<td>ddrinq</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>DFP Round To FP Integer Without Inexact Quad</td>
</tr>
<tr>
<td>111011 ...... 1110 00001</td>
<td>Z23</td>
<td>1</td>
<td>209</td>
<td>drrdd[]</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>DFP Round To FP Integer With Inexact</td>
</tr>
<tr>
<td>111011 ...... 1110 00001</td>
<td>Z23</td>
<td>1</td>
<td>209</td>
<td>drrddq</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>DFP Round To FP Integer With Inexact Quad</td>
</tr>
<tr>
<td>111011 ...... 000 00001</td>
<td>Z23</td>
<td>1</td>
<td>206</td>
<td>drrmd[]</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>DFP Reround</td>
</tr>
<tr>
<td>111011 ...... 000 00001</td>
<td>Z23</td>
<td>1</td>
<td>206</td>
<td>drrmdq</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>DFP Reround Quad</td>
</tr>
<tr>
<td>111011 ...... 1110 00001</td>
<td>X</td>
<td>1</td>
<td>214</td>
<td>dsfl[]</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>DFP Round To DFP Short</td>
</tr>
<tr>
<td>111011 ...... 010 00001</td>
<td>Z22</td>
<td>1</td>
<td>220</td>
<td>dscr[]</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>DFP Shift Significand Left Immediate</td>
</tr>
<tr>
<td>111011 ...... 010 00001</td>
<td>Z22</td>
<td>1</td>
<td>220</td>
<td>dscrq</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>DFP Shift Significand Left Immediate Quad</td>
</tr>
<tr>
<td>111011 ...... 010 00001</td>
<td>Z22</td>
<td>1</td>
<td>220</td>
<td>dssr[]</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>DFP Shift Significand Right Immediate</td>
</tr>
<tr>
<td>111011 ...... 010 00001</td>
<td>Z22</td>
<td>1</td>
<td>220</td>
<td>dssrq</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>DFP Shift Significand Right Immediate Quad</td>
</tr>
<tr>
<td>111011 ...... 1000 00001</td>
<td>X</td>
<td>1</td>
<td>193</td>
<td>dsub[]</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>DFP Subtract</td>
</tr>
<tr>
<td>111011 ...... 1000 00001</td>
<td>X</td>
<td>1</td>
<td>193</td>
<td>dsubq</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>DFP Subtract Quad</td>
</tr>
<tr>
<td>111011 ...... 1110 00001</td>
<td>Z22</td>
<td>1</td>
<td>202</td>
<td>dsstc</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>DFP Test Data Class</td>
</tr>
<tr>
<td>111011 ...... 1110 00001</td>
<td>Z22</td>
<td>1</td>
<td>202</td>
<td>dsstcq</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>DFP Test Data Class Quad</td>
</tr>
<tr>
<td>111011 ...... 1110 00001</td>
<td>Z22</td>
<td>1</td>
<td>202</td>
<td>dsstdg</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>DFP Test Data Group</td>
</tr>
<tr>
<td>111011 ...... 1110 00001</td>
<td>Z22</td>
<td>1</td>
<td>202</td>
<td>dsstdgq</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>DFP Test Data Group Quad</td>
</tr>
<tr>
<td>111011 ...... 1010 00001</td>
<td>X</td>
<td>1</td>
<td>201</td>
<td>dstex</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>DFP Test Exponent</td>
</tr>
<tr>
<td>111011 ...... 1010 00001</td>
<td>X</td>
<td>1</td>
<td>201</td>
<td>dstexq</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>DFP Test Exponent Quad</td>
</tr>
<tr>
<td>111011 ...... 1010 00001</td>
<td>X</td>
<td>1</td>
<td>202</td>
<td>dstf</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>DFP Test Significance</td>
</tr>
<tr>
<td>111011 ...... 1010 00001</td>
<td>X</td>
<td>1</td>
<td>202</td>
<td>dstfq</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>DFP Test Significance Immediate</td>
</tr>
<tr>
<td>111011 ...... 1010 00001</td>
<td>X</td>
<td>1</td>
<td>202</td>
<td>dstf</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>DFP Test Significance Immediate Quad</td>
</tr>
<tr>
<td>111011 ...... 1111 10101</td>
<td>X</td>
<td>1</td>
<td>218</td>
<td>dxex[]</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>DFP Extract Exponent</td>
</tr>
<tr>
<td>111011 ...... 1111 10101</td>
<td>X</td>
<td>1</td>
<td>218</td>
<td>dxexq</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>DFP Extract Exponent Quad</td>
</tr>
<tr>
<td>111011 ...... 1111 11111</td>
<td>X</td>
<td>1</td>
<td>875</td>
<td>eieio</td>
<td>PPC</td>
<td></td>
<td></td>
<td>Enforce In-order Execution of I/O</td>
</tr>
<tr>
<td>111011 ...... 1100 11111</td>
<td>X</td>
<td>1</td>
<td>95</td>
<td>eqv[]</td>
<td>P1</td>
<td>SR</td>
<td></td>
<td>Equivalent</td>
</tr>
<tr>
<td>111011 ...... 1110 11101</td>
<td>X</td>
<td>1</td>
<td>96</td>
<td>extal[]</td>
<td>PPC</td>
<td>SR</td>
<td></td>
<td>Extend Sign Byte</td>
</tr>
<tr>
<td>111011 ...... 1110 11101</td>
<td>X</td>
<td>1</td>
<td>96</td>
<td>extah[]</td>
<td>P1</td>
<td>SR</td>
<td></td>
<td>Extend Sign Halfword</td>
</tr>
<tr>
<td>111011 ...... 1110 11111</td>
<td>X</td>
<td>1</td>
<td>99</td>
<td>extaw[]</td>
<td>PPC</td>
<td>SR</td>
<td></td>
<td>Extend Sign Word</td>
</tr>
<tr>
<td>111011 ...... 1111 11101</td>
<td>XS</td>
<td>1</td>
<td>110</td>
<td>extowsl[]</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>Extend Sign Word &amp; Shift Left Immediate</td>
</tr>
<tr>
<td>111011 ...... 1111 11111</td>
<td>A</td>
<td>1</td>
<td>150</td>
<td>fab[]</td>
<td>P1</td>
<td></td>
<td></td>
<td>Floating Absolute</td>
</tr>
<tr>
<td>111011 ...... 1111 11111</td>
<td>A</td>
<td>1</td>
<td>152</td>
<td>fabd[]</td>
<td>P1</td>
<td></td>
<td></td>
<td>Floating Add</td>
</tr>
<tr>
<td>111011 ...... 1111 11111</td>
<td>A</td>
<td>1</td>
<td>152</td>
<td>fabds[]</td>
<td>PPC</td>
<td></td>
<td></td>
<td>Floating Add Single</td>
</tr>
<tr>
<td>111011 ...... 1111 11111</td>
<td>A</td>
<td>1</td>
<td>163</td>
<td>fcdl[]</td>
<td>PPC</td>
<td></td>
<td></td>
<td>Floating Convert with round Signed Doubleword to Double-Precision format</td>
</tr>
<tr>
<td>111011 ...... 1111 11111</td>
<td>A</td>
<td>1</td>
<td>164</td>
<td>fcdls[]</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>Floating Convert with round Signed Doubleword to Single-Precision format</td>
</tr>
<tr>
<td>111011 ...... 1111 11111</td>
<td>A</td>
<td>1</td>
<td>164</td>
<td>fcdlds[]</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>Floating Convert with round Unsigned Doubleword to Double-Precision format</td>
</tr>
<tr>
<td>111011 ...... 1111 11111</td>
<td>A</td>
<td>1</td>
<td>185</td>
<td>fcdfus[]</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>Floating Convert with round Unsigned Doubleword to Single-Precision format</td>
</tr>
<tr>
<td>111011 ...... 1110 11110</td>
<td>X</td>
<td>1</td>
<td>167</td>
<td>fcmpo</td>
<td>P1</td>
<td></td>
<td></td>
<td>Floating Compare Ordered</td>
</tr>
<tr>
<td>111011 ...... 1110 11110</td>
<td>X</td>
<td>1</td>
<td>167</td>
<td>fcmpu</td>
<td>P1</td>
<td></td>
<td></td>
<td>Floating Compare Unordered</td>
</tr>
<tr>
<td>111011 ...... 1110 10001</td>
<td>X</td>
<td>1</td>
<td>150</td>
<td>fcpsgn[]</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>Floating Copy Sign</td>
</tr>
<tr>
<td>111011 ...... 1110 10001</td>
<td>X</td>
<td>1</td>
<td>159</td>
<td>fcld[]</td>
<td>PPC</td>
<td></td>
<td></td>
<td>Floating Convert with round Double-Precision To Signed Doubleword format</td>
</tr>
<tr>
<td>111011 ...... 1110 10001</td>
<td>X</td>
<td>1</td>
<td>160</td>
<td>fcld[]</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>Floating Convert with round Double-Precision To Unsigned Doubleword format</td>
</tr>
<tr>
<td>111011 ...... 1110 10001</td>
<td>X</td>
<td>1</td>
<td>161</td>
<td>fcld[]</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>Floating Convert with round to Zero Double-Precision To Unsigned Doubleword format</td>
</tr>
<tr>
<td>111011 ...... 1110 10001</td>
<td>X</td>
<td>1</td>
<td>160</td>
<td>fcld[]</td>
<td>PPC</td>
<td></td>
<td></td>
<td>Floating Convert with round to Zero Double-Precision To Signed Doubleword format</td>
</tr>
</tbody>
</table>

Figure 90. Power ISA AS Instruction Set Sorted by Mnemonic (Sheet 3 of 18)
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>Book</th>
<th>Page</th>
<th>Mnemonic</th>
<th>Version2</th>
<th>Privilege3</th>
<th>Mode Dep4</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>X</td>
<td>I 161</td>
<td>fclix</td>
<td>P2</td>
<td>Floating Convert with round Double-Precision To Signed Word format</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
<td>I 162</td>
<td>fclix</td>
<td>v2.06</td>
<td>Floating Convert with round Double-Precision To Unsigned Word format</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
<td>I 163</td>
<td>fclix</td>
<td>v2.06</td>
<td>Floating Convert with round to Zero Double-Precision To Unsigned Word format</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
<td>I 162</td>
<td>fclix</td>
<td>P2</td>
<td>Floating Convert with round to Zero Double-Precision To Signed Word format</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>A</td>
<td>I 153</td>
<td>fdiv</td>
<td>P1</td>
<td>Floating Divide</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>A</td>
<td>I 153</td>
<td>fdiv</td>
<td>PPC</td>
<td>Floating Divide Single</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>A</td>
<td>I 157</td>
<td>fmadd</td>
<td>P1</td>
<td>Floating Multiply-Add</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>A</td>
<td>I 157</td>
<td>fmadd</td>
<td>PPC</td>
<td>Floating Multiply-Add Single</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>A</td>
<td>I 157</td>
<td>fmadds</td>
<td>PPC</td>
<td>Floating Negative Multiply-Add Single</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>A</td>
<td>I 158</td>
<td>fmsub</td>
<td>P1</td>
<td>Floating Multiply-Subtract</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>A</td>
<td>I 158</td>
<td>fmsubs</td>
<td>PPC</td>
<td>Floating Multiply-Subtract Single</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>A</td>
<td>I 153</td>
<td>fmul</td>
<td>P1</td>
<td>Floating Multiply</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>A</td>
<td>I 153</td>
<td>fmuls</td>
<td>PPC</td>
<td>Floating Multiply Single</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
<td>I 150</td>
<td>fnabs</td>
<td>P1</td>
<td>Floating Negative Absolute Value</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
<td>I 150</td>
<td>fneg</td>
<td>P1</td>
<td>Floating Negate</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>A</td>
<td>I 158</td>
<td>fnmadd</td>
<td>P1</td>
<td>Floating Negative Multiply-Add</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>A</td>
<td>I 158</td>
<td>fnmadds</td>
<td>PPC</td>
<td>Floating Negative Multiply-Add Single</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>A</td>
<td>I 158</td>
<td>fmuls</td>
<td>P1</td>
<td>Floating Negative Multiply-Subtract</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>A</td>
<td>I 158</td>
<td>fmuls</td>
<td>PPC</td>
<td>Floating Negative Multiply-Subtract Single</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
<td>I 154</td>
<td>frcst</td>
<td>v2.02</td>
<td>Floating Reciprocal Estimate</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
<td>I 154</td>
<td>frcst</td>
<td>PPC</td>
<td>Floating Reciprocal Estimate Single</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
<td>I 156</td>
<td>frintr</td>
<td>v2.02</td>
<td>Floating Round To Integer Plus</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
<td>I 166</td>
<td>frintr</td>
<td>v2.02</td>
<td>Floating Round To Integer Nearest</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
<td>I 166</td>
<td>frintr</td>
<td>v2.02</td>
<td>Floating Round To Integer Minus</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
<td>I 166</td>
<td>frintr</td>
<td>v2.02</td>
<td>Floating Round To Integer</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
<td>I 159</td>
<td>frintr</td>
<td>P1</td>
<td>Floating Round to Single-Precision</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>A</td>
<td>I 155</td>
<td>frsqrt</td>
<td>PPC</td>
<td>Floating Reciprocal Square Root Estimate</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>A</td>
<td>I 155</td>
<td>frsqrt</td>
<td>v2.02</td>
<td>Floating Reciprocal Square Root Single Estimate</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>A</td>
<td>I 168</td>
<td>fsel</td>
<td>PPC</td>
<td>Floating Select</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>A</td>
<td>I 154</td>
<td>fsel</td>
<td>P2</td>
<td>Floating Square Root</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>A</td>
<td>I 154</td>
<td>fsel</td>
<td>PPC</td>
<td>Floating Square Root Single</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>A</td>
<td>I 152</td>
<td>fsub</td>
<td>P1</td>
<td>Floating Subtract</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>A</td>
<td>I 152</td>
<td>fsub</td>
<td>PPC</td>
<td>Floating Subtract Single</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
<td>I 156</td>
<td>fdiv</td>
<td>v2.06</td>
<td>Floating Test for software Divide</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
<td>I 156</td>
<td>fdiv</td>
<td>v2.06</td>
<td>Floating Test for software Square Root</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
<td>I 111</td>
<td>1H</td>
<td>v2.02</td>
<td>Instruction Cache Block Invalidate</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
<td>I 110</td>
<td>1H</td>
<td>v2.07</td>
<td>Instruction Cache Block Touch</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>A</td>
<td>I 11</td>
<td>v2.03</td>
<td>Integer Select</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>XL</td>
<td>I 863</td>
<td>1H</td>
<td>P1</td>
<td>Instruction Synchronize</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
<td>I 864</td>
<td>1H</td>
<td>v2.06</td>
<td>Load Byte And Reserve Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>D</td>
<td>I 48</td>
<td>1H</td>
<td>P1</td>
<td>Load Byte &amp; Zero</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
<td>I 966</td>
<td>1H</td>
<td>v2.05</td>
<td>Load Byte &amp; Zero Caching Inhibited Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>D</td>
<td>I 48</td>
<td>1H</td>
<td>P1</td>
<td>Load Byte &amp; Zero With Update</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
<td>I 48</td>
<td>1H</td>
<td>P1</td>
<td>Load Byte &amp; Zero With Update Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
<td>I 48</td>
<td>1H</td>
<td>P1</td>
<td>Load Byte &amp; Zero Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>DS</td>
<td>I 53</td>
<td>0H</td>
<td>PPC</td>
<td>Load Doubleword</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
<td>I 869</td>
<td>1H</td>
<td>PPC</td>
<td>Load Doubleword And Reserve Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
<td>I 860</td>
<td>1H</td>
<td>v3.0</td>
<td>Load Doubleword Atomic</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
<td>I 61</td>
<td>1H</td>
<td>v2.06</td>
<td>Load Doubleword Byte-Reverse Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
<td>I 966</td>
<td>1H</td>
<td>v2.05</td>
<td>Load Doubleword Caching Inhibited Indexed</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 90. Power ISA AS Instruction Set Sorted by Mnemonic (Sheet 4 of 18)
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>Book</th>
<th>Page</th>
<th>Mnemonic</th>
<th>Version</th>
<th>Privilege</th>
<th>Mode</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111001</td>
<td>DS</td>
<td>I</td>
<td>53</td>
<td>ldu</td>
<td>PPC</td>
<td></td>
<td></td>
<td>Load Doubleword with Update</td>
</tr>
<tr>
<td>1111111</td>
<td>DS</td>
<td>I</td>
<td>53</td>
<td>ldx</td>
<td>PPC</td>
<td></td>
<td></td>
<td>Load Doubleword Indexed</td>
</tr>
<tr>
<td>1100001</td>
<td>D</td>
<td>I</td>
<td>142</td>
<td>fd</td>
<td>P1</td>
<td></td>
<td></td>
<td>Load Floating Double</td>
</tr>
<tr>
<td>1100111</td>
<td>D</td>
<td>I</td>
<td>142</td>
<td>fdp</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>Load Floating Double Pair</td>
</tr>
<tr>
<td>1100011</td>
<td>X</td>
<td>I</td>
<td>143</td>
<td>fidx</td>
<td>P1</td>
<td></td>
<td></td>
<td>Load Floating Double Indexed</td>
</tr>
<tr>
<td>1100101</td>
<td>X</td>
<td>I</td>
<td>143</td>
<td>fdpix</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>Load Floating as Integer Word Algebraic Indexed</td>
</tr>
<tr>
<td>1100001</td>
<td>D</td>
<td>I</td>
<td>141</td>
<td>fdu</td>
<td>P1</td>
<td></td>
<td></td>
<td>Load Floating Single</td>
</tr>
<tr>
<td>1100111</td>
<td>D</td>
<td>I</td>
<td>142</td>
<td>fdux</td>
<td>P1</td>
<td></td>
<td></td>
<td>Load Floating Single Indexed</td>
</tr>
<tr>
<td>1110101</td>
<td>D</td>
<td>I</td>
<td>50</td>
<td>fha</td>
<td>P1</td>
<td></td>
<td></td>
<td>Load Halfword Algebraic</td>
</tr>
<tr>
<td>1111101</td>
<td>X</td>
<td>I</td>
<td>865</td>
<td>fhanx</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>Load Halfword And Reserved Indexed Xform</td>
</tr>
<tr>
<td>1111101</td>
<td>X</td>
<td>I</td>
<td>50</td>
<td>fhau</td>
<td>P1</td>
<td></td>
<td></td>
<td>Load Halfword Algebraic with Update</td>
</tr>
<tr>
<td>1111101</td>
<td>X</td>
<td>I</td>
<td>50</td>
<td>fhaux</td>
<td>P1</td>
<td></td>
<td></td>
<td>Load Halfword Algebraic with Update Indexed</td>
</tr>
<tr>
<td>1100001</td>
<td>X</td>
<td>I</td>
<td>60</td>
<td>flbx</td>
<td>P1</td>
<td></td>
<td></td>
<td>Load Halfword Byte-Reverse Indexed</td>
</tr>
<tr>
<td>1101001</td>
<td>D</td>
<td>I</td>
<td>49</td>
<td>flz</td>
<td>P1</td>
<td></td>
<td></td>
<td>Load Halfword &amp; Zero</td>
</tr>
<tr>
<td>1100101</td>
<td>X</td>
<td>I</td>
<td>966</td>
<td>flczox</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>Load Halfword &amp; Zero Caching Inhibited Indexed</td>
</tr>
<tr>
<td>1100001</td>
<td>X</td>
<td>I</td>
<td>49</td>
<td>flczux</td>
<td>P1</td>
<td></td>
<td></td>
<td>Load Halfword &amp; Zero with Update Indexed</td>
</tr>
<tr>
<td>1110101</td>
<td>X</td>
<td>I</td>
<td>49</td>
<td>flczx</td>
<td>P1</td>
<td></td>
<td></td>
<td>Load Halfword &amp; Zero Indexed</td>
</tr>
<tr>
<td>1110001</td>
<td>D</td>
<td>I</td>
<td>62</td>
<td>fmx</td>
<td>P1</td>
<td></td>
<td></td>
<td>Load Multiple Word</td>
</tr>
<tr>
<td>1110001</td>
<td>D</td>
<td>I</td>
<td>58</td>
<td>flq</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>Load Quadword</td>
</tr>
<tr>
<td>1111101</td>
<td>X</td>
<td>I</td>
<td>871</td>
<td>fghax</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>Load Quadword And Reserved Indexed</td>
</tr>
<tr>
<td>1111101</td>
<td>X</td>
<td>I</td>
<td>64</td>
<td>fswi</td>
<td>P1</td>
<td></td>
<td></td>
<td>Load String Word Immediate</td>
</tr>
<tr>
<td>1111101</td>
<td>X</td>
<td>I</td>
<td>64</td>
<td>fswix</td>
<td>P1</td>
<td></td>
<td></td>
<td>Load String Word Indexed</td>
</tr>
<tr>
<td>1111101</td>
<td>X</td>
<td>I</td>
<td>242</td>
<td>fvebx</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>Load Vector Element Byte Indexed</td>
</tr>
<tr>
<td>1111101</td>
<td>X</td>
<td>I</td>
<td>242</td>
<td>fvehx</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>Load Vector Element Halfword Indexed</td>
</tr>
<tr>
<td>1111101</td>
<td>X</td>
<td>I</td>
<td>243</td>
<td>fvewx</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>Load Vector Element Word Indexed</td>
</tr>
<tr>
<td>1111101</td>
<td>X</td>
<td>I</td>
<td>247</td>
<td>fvlx</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>Load Vector for Shift Left</td>
</tr>
<tr>
<td>1111101</td>
<td>X</td>
<td>I</td>
<td>247</td>
<td>fvsr</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>Load Vector for Shift Right</td>
</tr>
<tr>
<td>1111101</td>
<td>X</td>
<td>I</td>
<td>245</td>
<td>fhx</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>Load Vector Indexed</td>
</tr>
<tr>
<td>1111101</td>
<td>X</td>
<td>I</td>
<td>245</td>
<td>fhxl</td>
<td>v2.03</td>
<td></td>
<td></td>
<td>Load Vector Indexed Last</td>
</tr>
<tr>
<td>1111101</td>
<td>X</td>
<td>I</td>
<td>50</td>
<td>fswi</td>
<td>PPC</td>
<td></td>
<td></td>
<td>Load Word Algebraic</td>
</tr>
<tr>
<td>1111101</td>
<td>X</td>
<td>I</td>
<td>865</td>
<td>fswix</td>
<td>PPC</td>
<td></td>
<td></td>
<td>Load Word &amp; Reserve Indexed</td>
</tr>
<tr>
<td>1111101</td>
<td>X</td>
<td>I</td>
<td>880</td>
<td>fswit</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>Load Word Atomic</td>
</tr>
<tr>
<td>1111101</td>
<td>X</td>
<td>I</td>
<td>52</td>
<td>fswux</td>
<td>PPC</td>
<td></td>
<td></td>
<td>Load Word Algebraic with Update Indexed</td>
</tr>
<tr>
<td>1111101</td>
<td>X</td>
<td>I</td>
<td>52</td>
<td>fswux</td>
<td>PPC</td>
<td></td>
<td></td>
<td>Load Word Algebraic Indexed</td>
</tr>
<tr>
<td>1111101</td>
<td>X</td>
<td>I</td>
<td>60</td>
<td>fswbx</td>
<td>P1</td>
<td></td>
<td></td>
<td>Load Word Byte-Reverse Indexed</td>
</tr>
<tr>
<td>1110101</td>
<td>D</td>
<td>I</td>
<td>51</td>
<td>fswz</td>
<td>P1</td>
<td></td>
<td></td>
<td>Load Word &amp; Zero</td>
</tr>
<tr>
<td>1111101</td>
<td>X</td>
<td>I</td>
<td>966</td>
<td>fszwzox</td>
<td>v2.05</td>
<td></td>
<td></td>
<td>Load Word &amp; Zero Caching Inhibited Indexed</td>
</tr>
<tr>
<td>1110101</td>
<td>D</td>
<td>I</td>
<td>51</td>
<td>fszwzux</td>
<td>P1</td>
<td></td>
<td></td>
<td>Load Word &amp; Zero with Update Indexed</td>
</tr>
<tr>
<td>1110101</td>
<td>X</td>
<td>I</td>
<td>51</td>
<td>fszwzux</td>
<td>P1</td>
<td></td>
<td></td>
<td>Load Word &amp; Zero Indexed</td>
</tr>
<tr>
<td>1110011</td>
<td>DS</td>
<td>I</td>
<td>480</td>
<td>ksdx</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>Load VSX Scalar Doubleword</td>
</tr>
<tr>
<td>1110011</td>
<td>X</td>
<td>I</td>
<td>480</td>
<td>ksdx</td>
<td>v2.06</td>
<td></td>
<td></td>
<td>Load VSX Scalar Doubleword Indexed</td>
</tr>
<tr>
<td>1110011</td>
<td>X</td>
<td>I</td>
<td>482</td>
<td>ksibzx</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>Load VSX Scalar as Integer Byte &amp; Zero Indexed</td>
</tr>
<tr>
<td>1110011</td>
<td>X</td>
<td>I</td>
<td>482</td>
<td>ksibzx</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>Load VSX Scalar as Integer Halfword &amp; Zero Indexed</td>
</tr>
<tr>
<td>1110011</td>
<td>X</td>
<td>I</td>
<td>483</td>
<td>ksibwx</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>Load VSX Scalar as Integer Word Indexed</td>
</tr>
<tr>
<td>1110011</td>
<td>X</td>
<td>I</td>
<td>483</td>
<td>ksibwx</td>
<td>v2.07</td>
<td></td>
<td></td>
<td>Load VSX Scalar as Integer Word &amp; Zero Indexed</td>
</tr>
<tr>
<td>1110011</td>
<td>X</td>
<td>I</td>
<td>485</td>
<td>kssp</td>
<td>v3.0</td>
<td></td>
<td></td>
<td>Load VSX Scalar Single</td>
</tr>
</tbody>
</table>

Figure 90. Power ISA AS Instruction Set Sorted by Mnemonic (Sheet 5 of 18)
<table>
<thead>
<tr>
<th>Instruction¹</th>
<th>Format</th>
<th>Book</th>
<th>Page</th>
<th>Mnemonic</th>
<th>Version²</th>
<th>Privilege³</th>
<th>Mode Dep⁴</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>011100 0000 0000</td>
<td>X</td>
<td>485</td>
<td>lvsspx</td>
<td>v2.07</td>
<td>Load VSX Scalar Single-Precision Indexed</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011101 0000 0000</td>
<td>X</td>
<td>492</td>
<td>lv</td>
<td>v3.0</td>
<td>Load VSX Vector</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011100 1110 0000</td>
<td>X</td>
<td>487</td>
<td>lvvb16x</td>
<td>v3.0</td>
<td>Load VSX Vector Byte*16 Indexed</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011100 1110 0000</td>
<td>X</td>
<td>488</td>
<td>lvvd2x</td>
<td>v2.06</td>
<td>Load VSX Vector Doubleword*2 Indexed</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011100 1110 0000</td>
<td>X</td>
<td>494</td>
<td>lvtxsx</td>
<td>v2.06</td>
<td>Load VSX Vector Doubleword &amp; Splat Indexed</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011100 1110 0000</td>
<td>X</td>
<td>495</td>
<td>lvxhlsx</td>
<td>v3.0</td>
<td>Load VSX Vector Halfword*8 Indexed</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011100 1110 0000</td>
<td>X</td>
<td>489</td>
<td>lvxl</td>
<td>v3.0</td>
<td>Load VSX Vector with Length</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011100 1110 0000</td>
<td>X</td>
<td>491</td>
<td>lvxil</td>
<td>v3.0</td>
<td>Load VSX Vector Left-justified with Length</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011100 1110 0000</td>
<td>X</td>
<td>496</td>
<td>lvxw4x</td>
<td>v2.06</td>
<td>Load VSX Vector Word*4 Indexed</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011100 1110 0000</td>
<td>X</td>
<td>497</td>
<td>lvxwsx</td>
<td>v3.0</td>
<td>Load VSX Vector Word &amp; Splat Indexed</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011100 1110 0000</td>
<td>X</td>
<td>492</td>
<td>lvx</td>
<td>v3.0</td>
<td>Load VSX Vector Indexed</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>013060 1110 0100</td>
<td>VA</td>
<td>80</td>
<td>maddhdd</td>
<td>v3.0</td>
<td>Multiply-Add High Doubleword</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>013060 1110 0100</td>
<td>VA</td>
<td>80</td>
<td>maddhudd</td>
<td>v3.0</td>
<td>Multiply-Add High Doubleword Unsigned</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>013060 1110 0100</td>
<td>VA</td>
<td>80</td>
<td>maddid</td>
<td>v3.0</td>
<td>Multiply-Add Low Doubleword</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>013060 1110 0100</td>
<td>X</td>
<td>41</td>
<td>mcfr</td>
<td>P1</td>
<td>Move CR Field</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>013060 1110 0100</td>
<td>X</td>
<td>909</td>
<td>mfbinrbe</td>
<td>v2.07</td>
<td>Move From BHRB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>013060 0000 0000</td>
<td>X</td>
<td>122</td>
<td>mtcr</td>
<td>P1</td>
<td>Move From CR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>013060 1000 0111</td>
<td>X</td>
<td>170</td>
<td>mts[.]</td>
<td>P1</td>
<td>Move From FPSCR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>013060 1010 0111</td>
<td>X</td>
<td>170</td>
<td>mtsfdrn</td>
<td>v3.0B</td>
<td>Move From FPSCR Control &amp; set DRN</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>013060 1010 0111</td>
<td>X</td>
<td>170</td>
<td>mtsfdrn</td>
<td>v3.0B</td>
<td>Move From FPSCR Control &amp; set DRN Immediate</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>013060 1000 0111</td>
<td>X</td>
<td>170</td>
<td>mtsfscr</td>
<td>v3.0B</td>
<td>Move From FPSCR Control &amp; set RN</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>013060 1000 0111</td>
<td>X</td>
<td>170</td>
<td>mtsfmsr</td>
<td>v3.0B</td>
<td>Move From FPSCR Control &amp; set RN Immediate</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>013060 1110 0111</td>
<td>X</td>
<td>170</td>
<td>mtsfmsr</td>
<td>v3.0B</td>
<td>Move From FPSCR Control &amp; set RN</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>013060 1110 0111</td>
<td>X</td>
<td>170</td>
<td>mtsfmsr</td>
<td>v3.0B</td>
<td>Move From FPSCR Control &amp; set RN Immediate</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>013060 1110 0111</td>
<td>X</td>
<td>170</td>
<td>mtsfl</td>
<td>v3.0B</td>
<td>Move From FPSCR Lightweight</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>013060 0000 0111</td>
<td>X</td>
<td>119</td>
<td>mfmsr</td>
<td>P1</td>
<td>Move From MSR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>013060 0000 0111</td>
<td>X</td>
<td>975</td>
<td>mfmsr</td>
<td>O</td>
<td>Move From MSR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>013060 0000 0111</td>
<td>X</td>
<td>898</td>
<td>mtsb</td>
<td>PPC</td>
<td>Move From Time Base</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>013060 1100 0110</td>
<td>VX</td>
<td>362</td>
<td>mtsvrd</td>
<td>v2.03</td>
<td>Move From VSCR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>013060 1100 0110</td>
<td>VX</td>
<td>112</td>
<td>mtsvrd</td>
<td>v2.07</td>
<td>Move From VSR Doubleword</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>013060 1100 0110</td>
<td>VX</td>
<td>112</td>
<td>mtsvrd</td>
<td>v3.0</td>
<td>Move From VSR Lower Doubleword</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>013060 1100 0110</td>
<td>VX</td>
<td>113</td>
<td>mtsvrdz</td>
<td>v2.07</td>
<td>Move From VSR Word &amp; Zero</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>013060 1100 0110</td>
<td>X</td>
<td>83</td>
<td>modsd</td>
<td>v3.0</td>
<td>Modulo Signed Doubleword</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>013060 1100 0110</td>
<td>X</td>
<td>77</td>
<td>modsw</td>
<td>v3.0</td>
<td>Modulo Signed Word</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>013060 1100 0110</td>
<td>X</td>
<td>83</td>
<td>mod</td>
<td>v3.0</td>
<td>Modulo Unsigned Doubleword</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>013060 1100 0110</td>
<td>X</td>
<td>77</td>
<td>modu</td>
<td>v3.0</td>
<td>Modulo Unsigned Word</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>013060 1100 0110</td>
<td>X</td>
<td>1130</td>
<td>msgdr</td>
<td>v2.07</td>
<td>HV</td>
<td>Message Clear</td>
<td></td>
<td></td>
</tr>
<tr>
<td>013060 1100 0110</td>
<td>X</td>
<td>1132</td>
<td>msgdrp</td>
<td>v2.07</td>
<td>P</td>
<td>Message Clear Privileged</td>
<td></td>
<td></td>
</tr>
<tr>
<td>013060 1100 0110</td>
<td>X</td>
<td>1129</td>
<td>msgsn</td>
<td>v2.07</td>
<td>HV</td>
<td>Message Send</td>
<td></td>
<td></td>
</tr>
<tr>
<td>013060 1100 0110</td>
<td>X</td>
<td>1131</td>
<td>msgndp</td>
<td>v2.07</td>
<td>P</td>
<td>Message Send Privileged</td>
<td></td>
<td></td>
</tr>
<tr>
<td>013060 1100 0110</td>
<td>X</td>
<td>1132</td>
<td>msgsync</td>
<td>v3.0</td>
<td>HV</td>
<td>Message Synchronize</td>
<td></td>
<td></td>
</tr>
<tr>
<td>013060 1100 0110</td>
<td>X</td>
<td>121</td>
<td>mtcrf</td>
<td>P1</td>
<td>Move To CR Fields</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>013060 1100 0110</td>
<td>X</td>
<td>173</td>
<td>mtfsd[.]</td>
<td>P1</td>
<td>Move To FPSCR Bit 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>013060 1100 0110</td>
<td>X</td>
<td>173</td>
<td>mtfsb[.]</td>
<td>P1</td>
<td>Move To FPSCR Bit 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>013060 1100 0110</td>
<td>X</td>
<td>172</td>
<td>mtfs[.]</td>
<td>P1</td>
<td>Move To FPSCR Fields</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>013060 1100 0110</td>
<td>X</td>
<td>172</td>
<td>mtfs[.]</td>
<td>P1</td>
<td>Move To FPSCR Field Immediate</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>013060 1100 0110</td>
<td>X</td>
<td>977</td>
<td>mtsmr</td>
<td>P1</td>
<td>Move To MSR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>013060 1100 0110</td>
<td>X</td>
<td>978</td>
<td>mtmsrd</td>
<td>P</td>
<td>Move To MSR Doubleword</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>013060 1100 0110</td>
<td>X</td>
<td>121</td>
<td>mtcrf</td>
<td>v2.01</td>
<td>Move To One CR Field</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>013060 1100 0110</td>
<td>X</td>
<td>117</td>
<td>mtsp</td>
<td>P1</td>
<td>Move To SPR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>013060 1100 0110</td>
<td>VX</td>
<td>362</td>
<td>mtvscr</td>
<td>v2.03</td>
<td>Move To VSCR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>013060 1100 0110</td>
<td>VX</td>
<td>114</td>
<td>mtvsvd</td>
<td>v2.07</td>
<td>Move To VSR Doubleword</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>013060 1100 0110</td>
<td>VX</td>
<td>115</td>
<td>mtsvzdd</td>
<td>v3.0</td>
<td>Move To VSR Double Doubleword</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 90. Power ISA AS Instruction Set Sorted by Mnemonic (Sheet 6 of 18)
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>Book</th>
<th>Page</th>
<th>Mnemonic</th>
<th>Version 2</th>
<th>Privilege</th>
<th>Mode Dep</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>XXI XXXX02</td>
<td>I</td>
<td>114</td>
<td>mtvrsra</td>
<td>v2.07</td>
<td>Move To VSR Word Algebraic</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XXI XXXX02</td>
<td>I</td>
<td>116</td>
<td>mtvrsrs</td>
<td>v3.0</td>
<td>Move To VSR Word &amp; Splat</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XXI XXXX02</td>
<td>I</td>
<td>115</td>
<td>mtvrsrwc</td>
<td>v2.07</td>
<td>Move To VSR Word &amp; Zero</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X XXXX01</td>
<td>I</td>
<td>79</td>
<td>mulh[ ]</td>
<td>PPC</td>
<td>SR Multiply High Doubleword</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X XXXX01</td>
<td>I</td>
<td>79</td>
<td>mulhd[ ]</td>
<td>PPC</td>
<td>SR Multiply High Doubleword Unsigned</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X XXXX01</td>
<td>I</td>
<td>73</td>
<td>mulh[ ]</td>
<td>PPC</td>
<td>SR Multiply High Word</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X XXXX01</td>
<td>I</td>
<td>73</td>
<td>mulhw[ ]</td>
<td>PPC</td>
<td>SR Multiply High Word Signed</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X XXXX01</td>
<td>I</td>
<td>79</td>
<td>mulh[ ]</td>
<td>PPC</td>
<td>SR Multiply Low Doubleword</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D XXXX01</td>
<td>I</td>
<td>73</td>
<td>muli</td>
<td>P1</td>
<td>Multiply Low Immediate</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X XXXX01</td>
<td>I</td>
<td>73</td>
<td>mul[ ]</td>
<td>P1</td>
<td>SR Multiply Low Word</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X XXXX01</td>
<td>I</td>
<td>94</td>
<td>nan[ ]</td>
<td>P1</td>
<td>SR NAND</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X XXXX01</td>
<td>I</td>
<td>94</td>
<td>neg[ ]</td>
<td>P1</td>
<td>SR Negate</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X XXXX01</td>
<td>I</td>
<td>95</td>
<td>nor[ ]</td>
<td>P1</td>
<td>SR NOR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X XXXX01</td>
<td>I</td>
<td>94</td>
<td>or[ ]</td>
<td>P1</td>
<td>SR OR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X XXXX01</td>
<td>I</td>
<td>95</td>
<td>orc[ ]</td>
<td>P1</td>
<td>SR OR with Complement</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D XXXX01</td>
<td>I</td>
<td>92</td>
<td>ori</td>
<td>P1</td>
<td>OR Immediate</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D XXXX01</td>
<td>I</td>
<td>93</td>
<td>oris</td>
<td>P1</td>
<td>OR Immediate Shifted</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X XXXX01</td>
<td>I</td>
<td>855</td>
<td>paste[ ]</td>
<td>v3.0</td>
<td>Paste</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X XXXX01</td>
<td>I</td>
<td>97</td>
<td>popcnb</td>
<td>v2.02</td>
<td>Population Count Byte</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X XXXX01</td>
<td>I</td>
<td>99</td>
<td>popcntb</td>
<td>v2.06</td>
<td>Population Count Doubleword</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X XXXX01</td>
<td>I</td>
<td>97</td>
<td>popcntw</td>
<td>v2.06</td>
<td>Population Count Words</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X XXXX01</td>
<td>I</td>
<td>98</td>
<td>prlyd</td>
<td>v2.05</td>
<td>Parity Doubleword</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X XXXX01</td>
<td>I</td>
<td>98</td>
<td>prlyw</td>
<td>v2.05</td>
<td>Parity Word</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XL XXXX01</td>
<td>I</td>
<td>905</td>
<td>retbo</td>
<td>v2.07</td>
<td>Return from Event Based Branch</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XL XXXX01</td>
<td>I</td>
<td>956</td>
<td>rdf</td>
<td>PPC</td>
<td>P</td>
<td>Return from Interrupt Doubleword</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XL XXXX01</td>
<td>I</td>
<td>953</td>
<td>rscv</td>
<td>v3.0</td>
<td>Return From System Call Vectored</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M XXXX01</td>
<td>I</td>
<td>104</td>
<td>rldcl</td>
<td>PPC</td>
<td>SR Rotate Left Doubleword then Clear Left</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M XXXX01</td>
<td>I</td>
<td>104</td>
<td>rldcr</td>
<td>PPC</td>
<td>SR Rotate Left Doubleword then Clear Right</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M XXXX01</td>
<td>I</td>
<td>105</td>
<td>rldcdl</td>
<td>PPC</td>
<td>SR Rotate Left Doubleword Immediate then Clear Left</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M XXXX01</td>
<td>I</td>
<td>105</td>
<td>rldc</td>
<td>PPC</td>
<td>SR Rotate Left Doubleword Immediate then Clear Right</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M XXXX01</td>
<td>I</td>
<td>106</td>
<td>rldc[ ]</td>
<td>PPC</td>
<td>SR Rotate Left Doubleword Immediate then Mask Insert</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M XXXX01</td>
<td>I</td>
<td>103</td>
<td>rldw[ ]</td>
<td>P1</td>
<td>SR Rotate Left Word Immediate then Mask Insert</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M XXXX01</td>
<td>I</td>
<td>102</td>
<td>rldw[ ]</td>
<td>P1</td>
<td>SR Rotate Left Word Immediate then AND with Mask</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M XXXX01</td>
<td>I</td>
<td>103</td>
<td>rldw[ ]</td>
<td>P1</td>
<td>SR Rotate Left Word Immediate then AND with Mask</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SC XXXX01</td>
<td>I</td>
<td>42</td>
<td>sc</td>
<td>PPC</td>
<td>System Call</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SC XXXX01</td>
<td>I</td>
<td>42</td>
<td>scv</td>
<td>v3.0</td>
<td>System Call Vectored</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>V2 XXXX01</td>
<td>I</td>
<td>122</td>
<td>setb</td>
<td>v3.0</td>
<td>Set Boolean</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XXI XXXX01</td>
<td>I</td>
<td>1031</td>
<td>sibflee</td>
<td>v2.05</td>
<td>SR SLB Find Entry ESB &amp; record</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XXI XXXX01</td>
<td>I</td>
<td>1026</td>
<td>sbia</td>
<td>PPC</td>
<td>P</td>
<td>SLB Invalidate All</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XXI XXXX01</td>
<td>I</td>
<td>1028</td>
<td>sibiag</td>
<td>v3.08</td>
<td>P</td>
<td>SLB Invalidate All Global</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XXI XXXX01</td>
<td>I</td>
<td>1024</td>
<td>sibie</td>
<td>PPC</td>
<td>P</td>
<td>SLB Invalidate Entry</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XXI XXXX01</td>
<td>I</td>
<td>1025</td>
<td>sibieg</td>
<td>v3.0</td>
<td>P</td>
<td>SLB Invalidate Entry Global</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XXI XXXX01</td>
<td>I</td>
<td>1031</td>
<td>sbmflee</td>
<td>v2.00</td>
<td>P</td>
<td>SLB Move From Entry ESI D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XXI XXXX01</td>
<td>I</td>
<td>1030</td>
<td>sbmfav</td>
<td>v2.00</td>
<td>P</td>
<td>SLB Move From Entry VSID</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XXI XXXX01</td>
<td>I</td>
<td>1029</td>
<td>sbmte</td>
<td>v2.00</td>
<td>P</td>
<td>SLB Move To Entry</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XXI XXXX01</td>
<td>I</td>
<td>1032</td>
<td>sbmrsync</td>
<td>v3.0</td>
<td>P</td>
<td>SLB Synchronize</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XXI XXXX01</td>
<td>I</td>
<td>109</td>
<td>sid[ ]</td>
<td>PPC</td>
<td>SR Shift Left Doubleword</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XXI XXXX01</td>
<td>I</td>
<td>107</td>
<td>siw[ ]</td>
<td>P1</td>
<td>SR Shift Left Word</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XXI XXXX01</td>
<td>I</td>
<td>110</td>
<td>siad[ ]</td>
<td>PPC</td>
<td>SR Shift Right Algebraic Doubleword</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XXI XXXX01</td>
<td>I</td>
<td>110</td>
<td>siadr</td>
<td>PPC</td>
<td>SR Shift Right Algebraic Doubleword Immediate</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XXI XXXX01</td>
<td>I</td>
<td>108</td>
<td>siaw</td>
<td>P1</td>
<td>SR Shift Right Algebraic Word</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XXI XXXX01</td>
<td>I</td>
<td>106</td>
<td>siaw[ ]</td>
<td>P1</td>
<td>SR Shift Right Algebraic Word Immediate</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XXI XXXX01</td>
<td>I</td>
<td>109</td>
<td>siord</td>
<td>PPC</td>
<td>SR Shift Right Doubleword</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X XXXX01</td>
<td>I</td>
<td>107</td>
<td>siw[ ]</td>
<td>P1</td>
<td>SR Shift Right Word</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X XXXX01</td>
<td>I</td>
<td>54</td>
<td>stb</td>
<td>P1</td>
<td>Store Byte</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 90. Power ISA AS Instruction Set Sorted by Mnemonic (Sheet 7 of 18)
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>Book</th>
<th>Page</th>
<th>Mnemonic</th>
<th>Version</th>
<th>Privilege</th>
<th>Mode Dep</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>I</td>
<td>I</td>
<td>sbtxc</td>
<td>v2.05</td>
<td>HV</td>
<td></td>
<td>Store Byte Caching Inhibited Indexed</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>I</td>
<td>I</td>
<td>sbtxc</td>
<td>v2.06</td>
<td>Store Byte Conditional Indexed &amp; record</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>D</td>
<td>I</td>
<td>P1</td>
<td>sbtu</td>
<td>P1</td>
<td>Store Byte with Update</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>I</td>
<td>I</td>
<td>sbd</td>
<td>P1</td>
<td>Store Byte with Update Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>I</td>
<td>I</td>
<td>sbdx</td>
<td>P1</td>
<td>Store Byte Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>DS</td>
<td>I</td>
<td>PPC</td>
<td>sid</td>
<td>PPC</td>
<td>Store Doubleword</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>I</td>
<td>I</td>
<td>stdat</td>
<td>v3.0</td>
<td>Store Doubleword Atomic</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>I</td>
<td>I</td>
<td>stdbrx</td>
<td>v2.06</td>
<td>Store Doubleword Byte-Reverse Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>I</td>
<td>I</td>
<td>stdbrx</td>
<td>v2.05</td>
<td>Store Doubleword Caching Inhibited Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>DS</td>
<td>I</td>
<td>PPC</td>
<td>stdc</td>
<td>P1</td>
<td>Store Doubleword Conditional Indexed &amp; record</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>I</td>
<td>I</td>
<td>stdc</td>
<td>PPC</td>
<td>Store Doubleword with Update</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>I</td>
<td>I</td>
<td>stdc</td>
<td>PPC</td>
<td>Store Doubleword with Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>I</td>
<td>I</td>
<td>stdc</td>
<td>PPC</td>
<td>Store Doubleword Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>D</td>
<td>I</td>
<td>P1</td>
<td>std</td>
<td>P1</td>
<td>Store Floating Double</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>DS</td>
<td>I</td>
<td>P2</td>
<td>stdp</td>
<td>v2.05</td>
<td>Store Floating Double Pair</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>I</td>
<td>I</td>
<td>stdpx</td>
<td>v2.05</td>
<td>Store Floating Double Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>D</td>
<td>I</td>
<td>P1</td>
<td>stdf</td>
<td>P1</td>
<td>Store Floating Double with Update</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>I</td>
<td>I</td>
<td>stdf</td>
<td>P1</td>
<td>Store Floating Double with Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>I</td>
<td>I</td>
<td>stdf</td>
<td>P1</td>
<td>Store Floating Double Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>D</td>
<td>I</td>
<td>P1</td>
<td>stdf</td>
<td>P1</td>
<td>Store Floating Single Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>D</td>
<td>I</td>
<td>P1</td>
<td>stds</td>
<td>P1</td>
<td>Store Floating Single</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>D</td>
<td>I</td>
<td>P1</td>
<td>stdsu</td>
<td>P1</td>
<td>Store Floating Single with Update</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>I</td>
<td>I</td>
<td>stds</td>
<td>P1</td>
<td>Store Floating Single with Update Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>I</td>
<td>I</td>
<td>stds</td>
<td>P1</td>
<td>Store Floating Single Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>D</td>
<td>I</td>
<td>P1</td>
<td>ssh</td>
<td>P1</td>
<td>Store Halfword</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>I</td>
<td>I</td>
<td>sshb</td>
<td>P1</td>
<td>Store Halfword Byte-Reverse Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>I</td>
<td>I</td>
<td>shx</td>
<td>v2.05</td>
<td>HV</td>
<td>Store Halfword Caching Inhibited Indexed</td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>I</td>
<td>I</td>
<td>shx</td>
<td>v2.06</td>
<td>Store Halfword Conditional Indexed &amp; record</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>D</td>
<td>I</td>
<td>P1</td>
<td>shu</td>
<td>P1</td>
<td>Store Halfword with Update</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>I</td>
<td>I</td>
<td>shux</td>
<td>P1</td>
<td>Store Halfword with Update Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>I</td>
<td>I</td>
<td>shx</td>
<td>P1</td>
<td>Store Halfword Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>D</td>
<td>I</td>
<td>P1</td>
<td>stm</td>
<td>P1</td>
<td>Store Multiple Word</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>XL</td>
<td>I</td>
<td>I</td>
<td>stop</td>
<td>v3.0</td>
<td>P</td>
<td>Stop</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DS</td>
<td>I</td>
<td>P1</td>
<td>stq</td>
<td>v2.03</td>
<td>Store Quadword</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>I</td>
<td>I</td>
<td>stqwc</td>
<td>v2.07</td>
<td>Store Quadword Conditional Indexed &amp; record</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>I</td>
<td>I</td>
<td>stswi</td>
<td>P1</td>
<td>Store String Word Immediate</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>I</td>
<td>I</td>
<td>stswix</td>
<td>P1</td>
<td>Store String Word Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>I</td>
<td>I</td>
<td>stvexbx</td>
<td>v2.03</td>
<td>Store Vector Element Byte Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>I</td>
<td>I</td>
<td>stvexbx</td>
<td>v2.03</td>
<td>Store Vector Element Halfword Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>I</td>
<td>I</td>
<td>stvexxx</td>
<td>v2.03</td>
<td>Store Vector Element Word Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>I</td>
<td>I</td>
<td>stvxx</td>
<td>v2.03</td>
<td>Store Vector Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>I</td>
<td>I</td>
<td>stvix</td>
<td>v2.03</td>
<td>Store Vector Indexed Last</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>D</td>
<td>I</td>
<td>P1</td>
<td>stw</td>
<td>P1</td>
<td>Store Word</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>I</td>
<td>I</td>
<td>stwbrx</td>
<td>P1</td>
<td>Store Word Byte-Reverse Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>I</td>
<td>I</td>
<td>stwbrx</td>
<td>v2.05</td>
<td>HV</td>
<td>Store Word Caching Inhibited Indexed</td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>I</td>
<td>I</td>
<td>stwsrc</td>
<td>PPC</td>
<td>Store Word Conditional Indexed &amp; record</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>D</td>
<td>I</td>
<td>P1</td>
<td>stwu</td>
<td>P1</td>
<td>Store Word with Update</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>I</td>
<td>I</td>
<td>stwux</td>
<td>P1</td>
<td>Store Word with Update Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>I</td>
<td>I</td>
<td>stwux</td>
<td>P1</td>
<td>Store Word Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>DS</td>
<td>I</td>
<td>P1</td>
<td>stx</td>
<td>v3.0</td>
<td>Store VSX Scalar Doubleword</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>I</td>
<td>I</td>
<td>stxsrc</td>
<td>v2.06</td>
<td>Store VSX Scalar Doubleword Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>I</td>
<td>I</td>
<td>stxsrc</td>
<td>v3.0</td>
<td>Store VSX Scalar as Integer Byte Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>I</td>
<td>I</td>
<td>stxsrc</td>
<td>v3.0</td>
<td>Store VSX Scalar as Integer Halfword Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>I</td>
<td>I</td>
<td>stxsrc</td>
<td>v3.0</td>
<td>Store VSX Scalar as Integer Word Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>I</td>
<td>I</td>
<td>stxsrc</td>
<td>v3.0</td>
<td>Store VSX Scalar Single-Precision</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 90. Power ISA AS Instruction Set Sorted by Mnemonic (Sheet 8 of 18)
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>Book</th>
<th>Page</th>
<th>Mnemonic</th>
<th>Version</th>
<th>Privilege</th>
<th>Mode Dep</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.5</td>
<td>6.10</td>
<td>11.15</td>
<td>16.20</td>
<td>21.15</td>
<td>26.31</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>502</td>
<td>v2.07</td>
<td>Store VSX Scalar Single-Precision Indexed</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>507</td>
<td>v3.0</td>
<td>Store VSX Vector</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>503</td>
<td>v3.0</td>
<td>Store VSX Vector Byte*16 Indexed</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>504</td>
<td>v2.06</td>
<td>Store VSX Vector Doubleword*2 Indexed</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>505</td>
<td>v3.0</td>
<td>Store VSX Vector Halfword*3 Indexed</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>501</td>
<td>v3.0</td>
<td>Store VSX Vector with Length</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>509</td>
<td>v3.0</td>
<td>Store VSX Vector Left-justified with Length</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>506</td>
<td>V2.00</td>
<td>Store VSX Vector Word*4 Indexed</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>510</td>
<td>v3.0</td>
<td>Store VSX Vector Indexed</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>69</td>
<td>P1</td>
<td>SR Subtract From</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>70</td>
<td>P1</td>
<td>SR Subtract From Carrying</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>71</td>
<td>P1</td>
<td>SR Subtract From Extended</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>70</td>
<td>P1</td>
<td>SR Subtract From Immediate Carrying</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>71</td>
<td>P1</td>
<td>SR Subtract From Minus One Extended</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>72</td>
<td>P1</td>
<td>SR Subtract From Zero Extended</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>873</td>
<td>P1</td>
<td>Synchronize</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>892</td>
<td>v2.07</td>
<td>Transaction Abort &amp; record</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>894</td>
<td>v2.07</td>
<td>Transaction Abort Doubleword Conditional &amp; record</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>893</td>
<td>v2.07</td>
<td>Transaction Abort Doubleword Conditional Immediate &amp; record</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>893</td>
<td>v2.07</td>
<td>Transaction Abort Word Conditional &amp; record</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>893</td>
<td>v2.07</td>
<td>Transaction Abort Word Conditional Immediate &amp; record</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>890</td>
<td>v2.07</td>
<td>Transaction Begin &amp; record</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>896</td>
<td>V2.07</td>
<td>Transaction Check &amp; record</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>91</td>
<td>91</td>
<td>PPCC</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>91</td>
<td>91</td>
<td>PPCC</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>891</td>
<td>v2.07</td>
<td>Transaction End &amp; record</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1043</td>
<td>P1</td>
<td>HV 64</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1038</td>
<td>P2</td>
<td>64</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1042</td>
<td>PPC</td>
<td>HV/P</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>970</td>
<td>v2.07</td>
<td>Transaction Recheckpoint &amp; record</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>969</td>
<td>v2.07</td>
<td>Transaction Reclaim &amp; record</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>896</td>
<td>v2.07</td>
<td>Transaction Suspend or Resume &amp; record</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>90</td>
<td>P1</td>
<td>Trap Word</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>90</td>
<td>P1</td>
<td>Trap Word Immediate</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>297</td>
<td>v3.0</td>
<td>Vector Absolute Difference Unsigned Byte</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>297</td>
<td>v3.0</td>
<td>Vector Absolute Difference Unsigned Halfword</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>296</td>
<td>v3.0</td>
<td>Vector Absolute Difference Unsigned Word</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>296</td>
<td>v3.0</td>
<td>Vector Absolute Difference Unsigned Word</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>273</td>
<td>V2.07</td>
<td>Vector Add &amp; write Carry Unsigned Quadrword</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>297</td>
<td>V2.07</td>
<td>Vector Add &amp; write Carry-OUT Unsigned Word</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>273</td>
<td>V2.07</td>
<td>Vector Add Extended &amp; write Carry Unsigned Quadrword</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>273</td>
<td>V2.07</td>
<td>Vector Add Extended Unsigned Quadrword Modulo</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>321</td>
<td>V2.03</td>
<td>Vector Add Floating-Point</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>269</td>
<td>V2.03</td>
<td>Vector Add Signed Byte Saturate</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>269</td>
<td>V2.03</td>
<td>Vector Add Signed Halfword Saturate</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>270</td>
<td>V2.03</td>
<td>Vector Add Signed Word Saturate</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>270</td>
<td>V2.03</td>
<td>Vector Add Unsigned Byte Modulo</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>272</td>
<td>V2.03</td>
<td>Vector Add Unsigned Byte Saturate</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>271</td>
<td>V2.03</td>
<td>Vector Add Unsigned Doubleword Modulo</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>272</td>
<td>V2.03</td>
<td>Vector Add Unsigned Halfword Saturate</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>270</td>
<td>V2.07</td>
<td>Vector Add Unsigned Quadword Modulo</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>271</td>
<td>V2.03</td>
<td>Vector Add Unsigned Word Saturate</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>272</td>
<td>V2.03</td>
<td>Vector Add Unsigned Word Saturate</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>312</td>
<td>V2.03</td>
<td>Vector Logical AND</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>312</td>
<td>V2.03</td>
<td>Vector Logical AND with Complement</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>295</td>
<td>V2.03</td>
<td>Vector Average Signed Byte</td>
</tr>
</tbody>
</table>

Figure 90. Power ISA AS Instruction Set Sorted by Mnemonic (Sheet 9 of 18)
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>Book</th>
<th>Page</th>
<th>Mnemonic</th>
<th>Privilege3</th>
<th>Mode Dep4</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>001000</td>
<td>000100</td>
<td>VX</td>
<td>295</td>
<td>vavghsh</td>
<td>v2.03</td>
<td></td>
<td>Vector Average Signed Halfword</td>
</tr>
<tr>
<td>001000</td>
<td>000100</td>
<td>VX</td>
<td>295</td>
<td>vavgsrk</td>
<td>v2.03</td>
<td></td>
<td>Vector Average Signed Word</td>
</tr>
<tr>
<td>001000</td>
<td>000000</td>
<td>VX</td>
<td>296</td>
<td>vavgbh</td>
<td>v2.03</td>
<td></td>
<td>Vector Average Unsigned Byte</td>
</tr>
<tr>
<td>001000</td>
<td>000000</td>
<td>VX</td>
<td>296</td>
<td>vavgh</td>
<td>v2.03</td>
<td></td>
<td>Vector Average Unsigned Halfword</td>
</tr>
<tr>
<td>001000</td>
<td>000000</td>
<td>VX</td>
<td>296</td>
<td>vavglw</td>
<td>v2.03</td>
<td></td>
<td>Vector Average Unsigned Word</td>
</tr>
<tr>
<td>001000</td>
<td>000100</td>
<td>VX</td>
<td>346</td>
<td>vxtpermcl</td>
<td>v3.0</td>
<td></td>
<td>Vector Bit Permute Doubleword</td>
</tr>
<tr>
<td>001000</td>
<td>001000</td>
<td>VX</td>
<td>346</td>
<td>vxtpermq</td>
<td>v2.07</td>
<td></td>
<td>Vector Bit Permute Quadword</td>
</tr>
<tr>
<td>001000</td>
<td>000010</td>
<td>VX</td>
<td>325</td>
<td>vextxx</td>
<td>v2.03</td>
<td></td>
<td>Vector Convert with round to nearest Signed Word format to FP</td>
</tr>
<tr>
<td>001000</td>
<td>000010</td>
<td>VX</td>
<td>325</td>
<td>vextxxs</td>
<td>v2.03</td>
<td></td>
<td>Vector Convert with round to nearest Signed Word format to FP</td>
</tr>
<tr>
<td>001000</td>
<td>001000</td>
<td>VX</td>
<td>333</td>
<td>vctx</td>
<td>v2.07</td>
<td></td>
<td>Vector AES Cipher</td>
</tr>
<tr>
<td>001000</td>
<td>001000</td>
<td>VX</td>
<td>333</td>
<td>vctxlast</td>
<td>v2.07</td>
<td></td>
<td>Vector AES Cipher Last</td>
</tr>
<tr>
<td>001000</td>
<td>000000</td>
<td>VX</td>
<td>340</td>
<td>vctxb</td>
<td>v2.07</td>
<td></td>
<td>Vector Count Leading Zeros Byte</td>
</tr>
<tr>
<td>001000</td>
<td>000000</td>
<td>VX</td>
<td>340</td>
<td>vctxd</td>
<td>v2.07</td>
<td></td>
<td>Vector Count Leading Zeros Doubleword</td>
</tr>
<tr>
<td>001000</td>
<td>000000</td>
<td>VX</td>
<td>340</td>
<td>vctxh</td>
<td>v2.07</td>
<td></td>
<td>Vector Count Leading Zeros Halfword</td>
</tr>
<tr>
<td>001000</td>
<td>000000</td>
<td>VX</td>
<td>340</td>
<td>vctxz</td>
<td>v2.07</td>
<td></td>
<td>Vector Count Leading Zeros Word</td>
</tr>
<tr>
<td>001000</td>
<td>000000</td>
<td>VX</td>
<td>340</td>
<td>vctxzlll</td>
<td>v2.07</td>
<td></td>
<td>Vector Count Leading Zeros Least-Significant Bits Word</td>
</tr>
<tr>
<td>001000</td>
<td>000000</td>
<td>VX</td>
<td>340</td>
<td>vctxzlll</td>
<td>v2.07</td>
<td></td>
<td>Vector Count Leading Zeros Least-Significant Bits Word</td>
</tr>
<tr>
<td>001000</td>
<td>000000</td>
<td>VX</td>
<td>340</td>
<td>vctxzlll</td>
<td>v2.07</td>
<td></td>
<td>Vector Count Leading Zeros Least-Significant Bits Word</td>
</tr>
<tr>
<td>001000</td>
<td>000000</td>
<td>VX</td>
<td>340</td>
<td>vctxzlll</td>
<td>v2.07</td>
<td></td>
<td>Vector Count Leading Zeros Least-Significant Bits Word</td>
</tr>
<tr>
<td>001000</td>
<td>000000</td>
<td>VX</td>
<td>340</td>
<td>vctxzlll</td>
<td>v2.07</td>
<td></td>
<td>Vector Count Leading Zeros Least-Significant Bits Word</td>
</tr>
</tbody>
</table>

Figure 90. Power ISA AS Instruction Set Sorted by Mnemonic (Sheet 10 of 18)
<table>
<thead>
<tr>
<th>Instruction¹</th>
<th>Format</th>
<th>Book</th>
<th>Page</th>
<th>Mnemonic</th>
<th>Version²</th>
<th>Privilege³</th>
<th>Mode Dep⁴</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>000100</td>
<td></td>
<td>0000100</td>
<td></td>
<td>VX</td>
<td>294</td>
<td>vextsh2w</td>
<td>v3.0</td>
</tr>
<tr>
<td></td>
<td>000100</td>
<td></td>
<td>0000100</td>
<td></td>
<td>VX</td>
<td>294</td>
<td>vextsw2d</td>
<td>v3.0</td>
</tr>
<tr>
<td></td>
<td>000100</td>
<td></td>
<td>0000100</td>
<td></td>
<td>VX</td>
<td>343</td>
<td>vextubix</td>
<td>v3.0</td>
</tr>
<tr>
<td></td>
<td>000100</td>
<td></td>
<td>0000100</td>
<td></td>
<td>VX</td>
<td>343</td>
<td>vextubrx</td>
<td>v3.0</td>
</tr>
<tr>
<td></td>
<td>000100</td>
<td></td>
<td>0000100</td>
<td></td>
<td>VX</td>
<td>343</td>
<td>vextuhix</td>
<td>v3.0</td>
</tr>
<tr>
<td></td>
<td>000100</td>
<td></td>
<td>0000100</td>
<td></td>
<td>VX</td>
<td>343</td>
<td>vextuhnx</td>
<td>v3.0</td>
</tr>
<tr>
<td></td>
<td>000100</td>
<td></td>
<td>0000100</td>
<td></td>
<td>VX</td>
<td>344</td>
<td>vextuwlk</td>
<td>v3.0</td>
</tr>
<tr>
<td></td>
<td>000100</td>
<td></td>
<td>0000100</td>
<td></td>
<td>VX</td>
<td>344</td>
<td>vextuwx</td>
<td>v3.0</td>
</tr>
<tr>
<td></td>
<td>000100</td>
<td></td>
<td>0000100</td>
<td></td>
<td>VX</td>
<td>338</td>
<td>vgbbd</td>
<td>v2.0</td>
</tr>
<tr>
<td></td>
<td>000100</td>
<td></td>
<td>0000100</td>
<td></td>
<td>VX</td>
<td>268</td>
<td>vinsertb</td>
<td>v3.0</td>
</tr>
<tr>
<td></td>
<td>000100</td>
<td></td>
<td>0000100</td>
<td></td>
<td>VX</td>
<td>268</td>
<td>vinsertd</td>
<td>v3.0</td>
</tr>
<tr>
<td></td>
<td>000100</td>
<td></td>
<td>0000100</td>
<td></td>
<td>VX</td>
<td>268</td>
<td>vinsertf</td>
<td>v3.0</td>
</tr>
<tr>
<td></td>
<td>000100</td>
<td></td>
<td>0000100</td>
<td></td>
<td>VX</td>
<td>268</td>
<td>vinsertw</td>
<td>v3.0</td>
</tr>
<tr>
<td></td>
<td>000100</td>
<td></td>
<td>0000100</td>
<td></td>
<td>VA</td>
<td>331</td>
<td>vlogefp</td>
<td>v2.0</td>
</tr>
<tr>
<td></td>
<td>000100</td>
<td></td>
<td>0000100</td>
<td></td>
<td>VX</td>
<td>322</td>
<td>vmaddfp</td>
<td>v2.0</td>
</tr>
<tr>
<td></td>
<td>000100</td>
<td></td>
<td>0000100</td>
<td></td>
<td>VX</td>
<td>323</td>
<td>vmaxfp</td>
<td>v2.0</td>
</tr>
<tr>
<td></td>
<td>000100</td>
<td></td>
<td>0000100</td>
<td></td>
<td>VX</td>
<td>299</td>
<td>vmaxsb</td>
<td>v2.0</td>
</tr>
<tr>
<td></td>
<td>000100</td>
<td></td>
<td>0000100</td>
<td></td>
<td>VX</td>
<td>299</td>
<td>vmaxsd</td>
<td>v2.0</td>
</tr>
<tr>
<td></td>
<td>000100</td>
<td></td>
<td>0000100</td>
<td></td>
<td>VX</td>
<td>300</td>
<td>vmaxsh</td>
<td>v2.0</td>
</tr>
<tr>
<td></td>
<td>000100</td>
<td></td>
<td>0000100</td>
<td></td>
<td>VX</td>
<td>300</td>
<td>vmaxsw</td>
<td>v2.0</td>
</tr>
<tr>
<td></td>
<td>000100</td>
<td></td>
<td>0000100</td>
<td></td>
<td>VX</td>
<td>299</td>
<td>vmaxxb</td>
<td>v2.0</td>
</tr>
<tr>
<td></td>
<td>000100</td>
<td></td>
<td>0000100</td>
<td></td>
<td>VX</td>
<td>299</td>
<td>vmaxxd</td>
<td>v2.0</td>
</tr>
<tr>
<td></td>
<td>000100</td>
<td></td>
<td>0000100</td>
<td></td>
<td>VX</td>
<td>300</td>
<td>vmaxux</td>
<td>v2.0</td>
</tr>
<tr>
<td></td>
<td>000100</td>
<td></td>
<td>0000100</td>
<td></td>
<td>VX</td>
<td>300</td>
<td>vmaxuxw</td>
<td>v2.0</td>
</tr>
<tr>
<td></td>
<td>000100</td>
<td></td>
<td>0000100</td>
<td></td>
<td>VA</td>
<td>285</td>
<td>vhmaddshs</td>
<td>v2.0</td>
</tr>
<tr>
<td></td>
<td>000100</td>
<td></td>
<td>0000100</td>
<td></td>
<td>VA</td>
<td>285</td>
<td>vhmradshs</td>
<td>v2.0</td>
</tr>
<tr>
<td></td>
<td>000100</td>
<td></td>
<td>0000100</td>
<td></td>
<td>VX</td>
<td>323</td>
<td>vminfp</td>
<td>v2.0</td>
</tr>
<tr>
<td></td>
<td>000100</td>
<td></td>
<td>0000100</td>
<td></td>
<td>VX</td>
<td>301</td>
<td>vminsb</td>
<td>v2.0</td>
</tr>
<tr>
<td></td>
<td>000100</td>
<td></td>
<td>0000100</td>
<td></td>
<td>VX</td>
<td>301</td>
<td>vminsd</td>
<td>v2.0</td>
</tr>
<tr>
<td></td>
<td>000100</td>
<td></td>
<td>0000100</td>
<td></td>
<td>VX</td>
<td>302</td>
<td>vminsh</td>
<td>v2.0</td>
</tr>
<tr>
<td></td>
<td>000100</td>
<td></td>
<td>0000100</td>
<td></td>
<td>VX</td>
<td>302</td>
<td>vminsw</td>
<td>v2.0</td>
</tr>
<tr>
<td></td>
<td>000100</td>
<td></td>
<td>0000100</td>
<td></td>
<td>VX</td>
<td>301</td>
<td>vminub</td>
<td>v2.0</td>
</tr>
<tr>
<td></td>
<td>000100</td>
<td></td>
<td>0000100</td>
<td></td>
<td>VX</td>
<td>301</td>
<td>vminud</td>
<td>v2.0</td>
</tr>
<tr>
<td></td>
<td>000100</td>
<td></td>
<td>0000100</td>
<td></td>
<td>VX</td>
<td>302</td>
<td>vminuh</td>
<td>v2.0</td>
</tr>
<tr>
<td></td>
<td>000100</td>
<td></td>
<td>0000100</td>
<td></td>
<td>VX</td>
<td>302</td>
<td>vminuw</td>
<td>v2.0</td>
</tr>
<tr>
<td></td>
<td>000100</td>
<td></td>
<td>0000100</td>
<td></td>
<td>VA</td>
<td>286</td>
<td>vmmadduhm</td>
<td>v2.0</td>
</tr>
<tr>
<td></td>
<td>000100</td>
<td></td>
<td>0000100</td>
<td></td>
<td>VX</td>
<td>257</td>
<td>vmmgew</td>
<td>v2.0</td>
</tr>
<tr>
<td></td>
<td>000100</td>
<td></td>
<td>0000100</td>
<td></td>
<td>VX</td>
<td>255</td>
<td>vmmghb</td>
<td>v2.0</td>
</tr>
<tr>
<td></td>
<td>000100</td>
<td></td>
<td>0000100</td>
<td></td>
<td>VX</td>
<td>255</td>
<td>vmmghh</td>
<td>v2.0</td>
</tr>
<tr>
<td></td>
<td>000100</td>
<td></td>
<td>0000100</td>
<td></td>
<td>VX</td>
<td>256</td>
<td>vmmghw</td>
<td>v2.0</td>
</tr>
<tr>
<td></td>
<td>000100</td>
<td></td>
<td>0000100</td>
<td></td>
<td>VX</td>
<td>255</td>
<td>vmmgb</td>
<td>v2.0</td>
</tr>
<tr>
<td></td>
<td>000100</td>
<td></td>
<td>0000100</td>
<td></td>
<td>VX</td>
<td>255</td>
<td>vmmgh</td>
<td>v2.0</td>
</tr>
<tr>
<td></td>
<td>000100</td>
<td></td>
<td>0000100</td>
<td></td>
<td>VX</td>
<td>256</td>
<td>vmmgw</td>
<td>v2.0</td>
</tr>
<tr>
<td></td>
<td>000100</td>
<td></td>
<td>0000100</td>
<td></td>
<td>VX</td>
<td>257</td>
<td>vmmgow</td>
<td>v2.0</td>
</tr>
<tr>
<td></td>
<td>000100</td>
<td></td>
<td>0000100</td>
<td></td>
<td>VA</td>
<td>287</td>
<td>vmsummbm</td>
<td>v2.0</td>
</tr>
<tr>
<td></td>
<td>000100</td>
<td></td>
<td>0000100</td>
<td></td>
<td>VA</td>
<td>287</td>
<td>vmsumshm</td>
<td>v2.0</td>
</tr>
<tr>
<td></td>
<td>000100</td>
<td></td>
<td>0000100</td>
<td></td>
<td>VA</td>
<td>288</td>
<td>vmsumshs</td>
<td>v2.0</td>
</tr>
<tr>
<td></td>
<td>000100</td>
<td></td>
<td>0000100</td>
<td></td>
<td>VA</td>
<td>286</td>
<td>vmsumubm</td>
<td>v2.0</td>
</tr>
<tr>
<td></td>
<td>000100</td>
<td></td>
<td>0000100</td>
<td></td>
<td>VA</td>
<td>289</td>
<td>vmsumudm</td>
<td>v3.0</td>
</tr>
<tr>
<td></td>
<td>000100</td>
<td></td>
<td>0000100</td>
<td></td>
<td>VA</td>
<td>289</td>
<td>vmsumumh</td>
<td>v2.0</td>
</tr>
<tr>
<td></td>
<td>000100</td>
<td></td>
<td>0000100</td>
<td></td>
<td>VA</td>
<td>289</td>
<td>vmsumuh</td>
<td>v2.0</td>
</tr>
<tr>
<td></td>
<td>000100</td>
<td></td>
<td>0000100</td>
<td></td>
<td>VX</td>
<td>355</td>
<td>vmu10ucq</td>
<td>v3.0</td>
</tr>
<tr>
<td></td>
<td>000100</td>
<td></td>
<td>0000100</td>
<td></td>
<td>VX</td>
<td>355</td>
<td>vmu10ecq</td>
<td>v3.0</td>
</tr>
<tr>
<td></td>
<td>000100</td>
<td></td>
<td>0000100</td>
<td></td>
<td>VX</td>
<td>355</td>
<td>vmu10cuq</td>
<td>v3.0</td>
</tr>
<tr>
<td></td>
<td>000100</td>
<td></td>
<td>0000100</td>
<td></td>
<td>VX</td>
<td>281</td>
<td>vmu1esb</td>
<td>v2.0</td>
</tr>
</tbody>
</table>

Figure 90. Power ISA AS Instruction Set Sorted by Mnemonic (Sheet 11 of 18)

Appendix F. Power ISA Instruction Set Sorted by Mnemonic 1229
### Power ISA™ Appendices

#### Version 3.0 B

<table>
<thead>
<tr>
<th>Instruction 1</th>
<th>Format</th>
<th>Book</th>
<th>Page</th>
<th>Mnemonic</th>
<th>Version2</th>
<th>Privilege3</th>
<th>Mode Dep4</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0:5</td>
<td>6:10</td>
<td>13:15</td>
<td>16:20</td>
<td>21:25</td>
<td>26:31</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>0181</td>
<td>00100</td>
<td>282</td>
<td>vmulesh</td>
<td>v2.03</td>
<td>Vector Multiply Even Signed Halfword</td>
<td></td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>0110</td>
<td>00100</td>
<td>283</td>
<td>vmulesw</td>
<td>v2.07</td>
<td>Vector Multiply Even Signed Word</td>
<td></td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>0100</td>
<td>00100</td>
<td>281</td>
<td>vmuleub</td>
<td>v2.03</td>
<td>Vector Multiply Even Unsigned Byte</td>
<td></td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>0101</td>
<td>00100</td>
<td>282</td>
<td>vmuleuh</td>
<td>v2.03</td>
<td>Vector Multiply Even Unsigned Halfword</td>
<td></td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>0110</td>
<td>00100</td>
<td>282</td>
<td>vmulesw</td>
<td>v2.07</td>
<td>Vector Multiply Even Signed Word</td>
<td></td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>0110</td>
<td>00100</td>
<td>281</td>
<td>vmulob</td>
<td>v2.03</td>
<td>Vector Multiply Odd Signed Byte</td>
<td></td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>0110</td>
<td>00100</td>
<td>282</td>
<td>vmulosh</td>
<td>v2.03</td>
<td>Vector Multiply Odd Signed Halfword</td>
<td></td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>0100</td>
<td>00100</td>
<td>283</td>
<td>vmulow</td>
<td>v2.07</td>
<td>Vector Multiply Odd Signed Word</td>
<td></td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>0100</td>
<td>00100</td>
<td>281</td>
<td>vmulub</td>
<td>v2.03</td>
<td>Vector Multiply Odd Unsigned Byte</td>
<td></td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>0100</td>
<td>00100</td>
<td>282</td>
<td>ymulesh</td>
<td>v2.03</td>
<td>Vector Multiply Odd Unsigned Halfword</td>
<td></td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>0100</td>
<td>00100</td>
<td>282</td>
<td>ymulew</td>
<td>v2.07</td>
<td>Vector Multiply Odd Signed Word</td>
<td></td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>0100</td>
<td>00100</td>
<td>281</td>
<td>ymulob</td>
<td>v2.03</td>
<td>Vector Multiply Odd Signed Byte</td>
<td></td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>0100</td>
<td>00100</td>
<td>282</td>
<td>ymulosh</td>
<td>v2.03</td>
<td>Vector Multiply Odd Signed Halfword</td>
<td></td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>0100</td>
<td>00100</td>
<td>283</td>
<td>ymulow</td>
<td>v2.07</td>
<td>Vector Multiply Odd Signed Word</td>
<td></td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>0100</td>
<td>00100</td>
<td>281</td>
<td>ymulub</td>
<td>v2.03</td>
<td>Vector Multiply Odd Unsigned Byte</td>
<td></td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>0100</td>
<td>00100</td>
<td>282</td>
<td>ymuluh</td>
<td>v2.03</td>
<td>Vector Multiply Odd Unsigned Halfword</td>
<td></td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>0100</td>
<td>00100</td>
<td>282</td>
<td>ymululw</td>
<td>v2.07</td>
<td>Vector Multiply Odd Signed Word</td>
<td></td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>0100</td>
<td>00100</td>
<td>281</td>
<td>ymululub</td>
<td>v2.03</td>
<td>Vector Multiply Odd Unsigned Byte</td>
<td></td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>0100</td>
<td>00100</td>
<td>282</td>
<td>ymululuh</td>
<td>v2.03</td>
<td>Vector Multiply Odd Unsigned Halfword</td>
<td></td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>0100</td>
<td>00100</td>
<td>282</td>
<td>ymulululw</td>
<td>v2.07</td>
<td>Vector Multiply Odd Signed Word</td>
<td></td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>0100</td>
<td>00100</td>
<td>281</td>
<td>ymulululub</td>
<td>v2.03</td>
<td>Vector Multiply Odd Unsigned Byte</td>
<td></td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>0100</td>
<td>00100</td>
<td>282</td>
<td>ymulululuh</td>
<td>v2.03</td>
<td>Vector Multiply Odd Unsigned Halfword</td>
<td></td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>0100</td>
<td>00100</td>
<td>282</td>
<td>ymululululw</td>
<td>v2.07</td>
<td>Vector Multiply Odd Signed Word</td>
<td></td>
</tr>
<tr>
<td>00100</td>
<td>VX</td>
<td>0100</td>
<td>00100</td>
<td>281</td>
<td>ymululululub</td>
<td>v2.03</td>
<td>Vector Multiply Odd Unsigned Byte</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 90.** Power ISA AS Instruction Set Sorted by Mnemonic (Sheet 12 of 18)
<table>
<thead>
<tr>
<th>Instruction¹</th>
<th>Format</th>
<th>Book</th>
<th>Page</th>
<th>Mnemonic</th>
<th>Version²</th>
<th>Privilege³</th>
<th>Mode Dep⁴</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>
</tr>
<tr>
<td>000100</td>
<td></td>
<td></td>
<td></td>
<td>vx</td>
<td>v3.0</td>
<td>B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td></td>
<td></td>
<td></td>
<td>vith</td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td></td>
<td></td>
<td></td>
<td>vrrw</td>
<td>v2.03</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td></td>
<td></td>
<td></td>
<td>vxrwm</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td></td>
<td></td>
<td></td>
<td>vxrwm</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td></td>
<td></td>
<td></td>
<td>vxrwm</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td></td>
<td></td>
<td></td>
<td>vxrwm</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td></td>
<td></td>
<td></td>
<td>vxrwm</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td></td>
<td></td>
<td></td>
<td>vxrwm</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td></td>
<td></td>
<td></td>
<td>vxrwm</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td></td>
<td></td>
<td></td>
<td>vxrwm</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td></td>
<td></td>
<td></td>
<td>vxrwm</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td></td>
<td></td>
<td></td>
<td>vxrwm</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td></td>
<td></td>
<td></td>
<td>vxrwm</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td></td>
<td></td>
<td></td>
<td>vxrwm</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td></td>
<td></td>
<td></td>
<td>vxrwm</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td></td>
<td></td>
<td></td>
<td>vxrwm</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td></td>
<td></td>
<td></td>
<td>vxrwm</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td></td>
<td></td>
<td></td>
<td>vxrwm</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td></td>
<td></td>
<td></td>
<td>vxrwm</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td></td>
<td></td>
<td></td>
<td>vxrwm</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td></td>
<td></td>
<td></td>
<td>vxrwm</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td></td>
<td></td>
<td></td>
<td>vxrwm</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td></td>
<td></td>
<td></td>
<td>vxrwm</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td></td>
<td></td>
<td></td>
<td>vxrwm</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td></td>
<td></td>
<td></td>
<td>vxrwm</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td></td>
<td></td>
<td></td>
<td>vxrwm</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td></td>
<td></td>
<td></td>
<td>vxrwm</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td></td>
<td></td>
<td></td>
<td>vxrwm</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td></td>
<td></td>
<td></td>
<td>vxrwm</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td></td>
<td></td>
<td></td>
<td>vxrwm</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td></td>
<td></td>
<td></td>
<td>vxrwm</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td></td>
<td></td>
<td></td>
<td>vxrwm</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td></td>
<td></td>
<td></td>
<td>vxrwm</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td></td>
<td></td>
<td></td>
<td>vxrwm</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td></td>
<td></td>
<td></td>
<td>vxrwm</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td></td>
<td></td>
<td></td>
<td>vxrwm</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td></td>
<td></td>
<td></td>
<td>vxrwm</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td></td>
<td></td>
<td></td>
<td>vxrwm</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td></td>
<td></td>
<td></td>
<td>vxrwm</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td></td>
<td></td>
<td></td>
<td>vxrwm</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td></td>
<td></td>
<td></td>
<td>vxrwm</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td></td>
<td></td>
<td></td>
<td>vxrwm</td>
<td>v3.0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 90. Power ISA AS Instruction Set Sorted by Mnemonic (Sheet 13 of 18)

Appendix F. Power ISA Instruction Set Sorted by Mnemonic 1231
<table>
<thead>
<tr>
<th>Instruction1</th>
<th>Format</th>
<th>Book</th>
<th>Page</th>
<th>Mnemonic</th>
<th>Version2</th>
<th>Privilege3</th>
<th>Mode Dep4</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>001000 001000 001010</td>
<td>VX I</td>
<td>253</td>
<td>VupkhpX</td>
<td>v2.03</td>
<td>Vector Unpack High Pixel</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001000 001010 001100</td>
<td>VX I</td>
<td>254</td>
<td>Vupkhsb</td>
<td>v2.03</td>
<td>Vector Unpack High Signed Byte</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001000 001100 001110</td>
<td>VX I</td>
<td>254</td>
<td>Vupksh</td>
<td>v2.03</td>
<td>Vector Unpack High Signed Halfword</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001000 001110 001111</td>
<td>VX I</td>
<td>254</td>
<td>Vupkhaw</td>
<td>v2.07</td>
<td>Vector Unpack High Signed Word</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001000 010000 010010</td>
<td>VX I</td>
<td>253</td>
<td>Vupkpx</td>
<td>v2.03</td>
<td>Vector Unpack Low Pixel</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001000 010010 010100</td>
<td>VX I</td>
<td>254</td>
<td>Vupklb</td>
<td>v2.03</td>
<td>Vector Unpack Low Signed Byte</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001000 010100 010110</td>
<td>VX I</td>
<td>254</td>
<td>Vupksh</td>
<td>v2.03</td>
<td>Vector Unpack Low Signed Halfword</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001000 010110 011000</td>
<td>VX I</td>
<td>254</td>
<td>Vupklow</td>
<td>v2.07</td>
<td>Vector Unpack Low Signed Word</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001000 011000 011010</td>
<td>VX I</td>
<td>313</td>
<td>Vxor</td>
<td>v2.03</td>
<td>Vector Logical XOR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001000 011010 011110</td>
<td>X</td>
<td>976</td>
<td>Wait</td>
<td>v3.0</td>
<td>Wait for Interrupt</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000000 000000 000000 000000 000000</td>
<td>D</td>
<td>93</td>
<td>Xnop</td>
<td>v2.05</td>
<td>Executed No Operation</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000000 000000 000000 000000 000000</td>
<td>X</td>
<td>94</td>
<td>Xor</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000000 000000 000000 000000 000000</td>
<td>D</td>
<td>93</td>
<td>Xori</td>
<td>P1</td>
<td>XOR Immediate</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000000 000000 000000 000000 000000</td>
<td>D</td>
<td>93</td>
<td>Xorib</td>
<td>P1</td>
<td>XOR Immediate Shifted</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001000 010000 010010</td>
<td>XX2</td>
<td>512</td>
<td>Xabsdp</td>
<td>v2.06</td>
<td>VSX Scalar Absolute Double-Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001000 110000 010100</td>
<td>X</td>
<td>512</td>
<td>Xabsbsp</td>
<td>v3.0</td>
<td>VSX Scalar Absolute Quad-Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001000 001000 001010</td>
<td>XX3</td>
<td>513</td>
<td>Xsadddp</td>
<td>v2.06</td>
<td>VSX Scalar Add Double-Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001000 000000 000010 000100 000110</td>
<td>X</td>
<td>520</td>
<td>Xsadddp[o]</td>
<td>v3.0</td>
<td>VSX Scalar Add Quad-Precision (with round to Odd)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001000 000000 000010 000100 000110</td>
<td>XX3</td>
<td>518</td>
<td>Xsadddp</td>
<td>v2.07</td>
<td>VSX Scalar Add Single-Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000000 000000 000010 000100 000110</td>
<td>XX3</td>
<td>524</td>
<td>Xscmpqdp</td>
<td>v3.0</td>
<td>VSX Scalar Compare Equal Double-Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000000 000100 000110 000110 000110</td>
<td>X</td>
<td>523</td>
<td>Xscmpqdp</td>
<td>v3.0</td>
<td>VSX Scalar Compare Exponents Double-Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000000 000100 000110 000110 000110</td>
<td>XX3</td>
<td>523</td>
<td>Xscmpqdp</td>
<td>v3.0</td>
<td>VSX Scalar Compare Exponents Quad-Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000000 000100 000110 000110 000110</td>
<td>XX3</td>
<td>527</td>
<td>Xscmpqdpp</td>
<td>v2.06</td>
<td>VSX Scalar Compare Ordered Double-Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000000 000100 000110 000110 000110</td>
<td>X</td>
<td>529</td>
<td>Xscmpqdp</td>
<td>v3.0</td>
<td>VSX Scalar Compare Ordered Quad-Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000000 000100 000110 000110 000110</td>
<td>XX3</td>
<td>530</td>
<td>Xscmpqdp</td>
<td>v2.06</td>
<td>VSX Scalar Compare Unordered Double-Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000000 000100 000110 000110 000110</td>
<td>X</td>
<td>532</td>
<td>Xscmpqdp</td>
<td>v3.0</td>
<td>VSX Scalar Compare Unordered Quad-Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000000 000100 000110 000110 000110</td>
<td>XX3</td>
<td>533</td>
<td>Xcpgsndp</td>
<td>v2.06</td>
<td>VSX Scalar Copy Sign Double-Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000000 000100 000110 000110 000110</td>
<td>XX3</td>
<td>533</td>
<td>Xcpgn ep</td>
<td>v3.0</td>
<td>VSX Scalar Copy Sign Quad-Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000000 000100 000110 000110 000110</td>
<td>X</td>
<td>534</td>
<td>Xscdpdp</td>
<td>v2.06</td>
<td>VSX Scalar Convert with round Double-Precision to Signed Doubleword format</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000000 000100 000110 000110 000110</td>
<td>XX2</td>
<td>536</td>
<td>Xscdpdp</td>
<td>v2.06</td>
<td>VSX Scalar Convert with round Double-Precision to Single-Precision format</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000000 000100 000110 000110 000110</td>
<td>XX2</td>
<td>537</td>
<td>Xscdpdsn</td>
<td>v2.07</td>
<td>VSX Scalar Convert Double-Precision to Single-Precision Non-signalling format</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000000 000100 000110 000110 000110</td>
<td>XX2</td>
<td>537</td>
<td>Xscdpdns</td>
<td>v2.06</td>
<td>VSX Scalar Convert with round to zero Double-Precision to Signed Doubleword format</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000000 000100 000110 000110 000110</td>
<td>XX2</td>
<td>540</td>
<td>Xscdpdxws</td>
<td>v2.06</td>
<td>VSX Scalar Convert with round to zero Double-Precision to Signed Word format</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000000 000100 000110 000110 000110</td>
<td>XX2</td>
<td>542</td>
<td>Xscdpdxwds</td>
<td>v2.06</td>
<td>VSX Scalar Convert with round to zero Double-Precision to Unsigned Doubleword format</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000000 000100 000110 000110 000110</td>
<td>XX2</td>
<td>544</td>
<td>Xscdpdxwes</td>
<td>v2.06</td>
<td>VSX Scalar Convert with round to zero Double-Precision to Unsigned Word format</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000000 000100 000110 000110 000110</td>
<td>XX2</td>
<td>546</td>
<td>Xscvdpdp</td>
<td>v3.0</td>
<td>VSX Scalar Convert Half-Precision to Double-Precision format</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000000 000100 000110 000110 000110</td>
<td>X</td>
<td>547</td>
<td>Xscvdpdp[o]</td>
<td>v3.0</td>
<td>VSX Scalar Convert with round Quad-Precision to Double-Precision format with round to Odd</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000000 000100 000110 000110 000110</td>
<td>X</td>
<td>548</td>
<td>Xscvgpsdz</td>
<td>v3.0</td>
<td>VSX Scalar Convert with round to zero Quad-Precision to Signed Doubleword format</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000000 000100 000110 000110 000110</td>
<td>X</td>
<td>550</td>
<td>Xscvgpswz</td>
<td>v3.0</td>
<td>VSX Scalar Convert with round to zero Quad-Precision to Signed Word format</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000000 000100 000110 000110 000110</td>
<td>X</td>
<td>552</td>
<td>Xscvgpudz</td>
<td>v3.0</td>
<td>VSX Scalar Convert with round to zero Quad-Precision to Unsigned Doubleword format</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000000 000100 000110 000110 000110</td>
<td>X</td>
<td>554</td>
<td>Xscvgpuzwz</td>
<td>v3.0</td>
<td>VSX Scalar Convert with round to zero Quad-Precision to Unsigned Word format</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000000 000100 000110 000110 000110</td>
<td>X</td>
<td>556</td>
<td>Xscvgdp</td>
<td>v3.0</td>
<td>VSX Scalar Convert Signed Doubleword to Quad-Precision format</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000000 000100 000110 000110 000110</td>
<td>XX2</td>
<td>557</td>
<td>Xscvgdp</td>
<td>v2.06</td>
<td>VSX Scalar Convert Single-Precision to Double-Precision format</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 90. Power ISA AS Instruction Set Sorted by Mnemonic (Sheet 14 of 18)
<table>
<thead>
<tr>
<th>Instruction1</th>
<th>Format</th>
<th>Book</th>
<th>Page</th>
<th>Mnemonic</th>
<th>Version</th>
<th>Privilege</th>
<th>Mode Dep</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x3100</td>
<td>0x1100</td>
<td>X3</td>
<td>1</td>
<td>xscvrdp</td>
<td>v2.07</td>
<td>VSX Scalar Convert Double-Precision</td>
<td>Non-signalling format</td>
<td></td>
</tr>
<tr>
<td>0x3100</td>
<td>0x1100</td>
<td>X3</td>
<td>1</td>
<td>xscvrdsp</td>
<td>v2.07</td>
<td>VSX Scalar Convert Double-Precision</td>
<td>Non-signalling format</td>
<td></td>
</tr>
<tr>
<td>0x3100</td>
<td>0x1100</td>
<td>X3</td>
<td>1</td>
<td>xscvredp</td>
<td>v2.07</td>
<td>VSX Scalar Convert Double-Precision</td>
<td>Non-signalling format</td>
<td></td>
</tr>
<tr>
<td>0x3100</td>
<td>0x1100</td>
<td>X3</td>
<td>1</td>
<td>xscvredsp</td>
<td>v2.07</td>
<td>VSX Scalar Convert Double-Precision</td>
<td>Non-signalling format</td>
<td></td>
</tr>
<tr>
<td>0x3100</td>
<td>0x1100</td>
<td>X3</td>
<td>1</td>
<td>xscvrdp</td>
<td>v2.07</td>
<td>VSX Scalar Convert Double-Precision</td>
<td>Non-signalling format</td>
<td></td>
</tr>
<tr>
<td>0x3100</td>
<td>0x1100</td>
<td>X3</td>
<td>1</td>
<td>xscvredp</td>
<td>v2.07</td>
<td>VSX Scalar Convert Double-Precision</td>
<td>Non-signalling format</td>
<td></td>
</tr>
<tr>
<td>0x3100</td>
<td>0x1100</td>
<td>X3</td>
<td>1</td>
<td>xscvredsp</td>
<td>v2.07</td>
<td>VSX Scalar Convert Double-Precision</td>
<td>Non-signalling format</td>
<td></td>
</tr>
</tbody>
</table>

Figure 90. Power ISA AS Instruction Set Sorted by Mnemonic (Sheet 15 of 18)
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>Book</th>
<th>Page</th>
<th>Mnemonic</th>
<th>Version</th>
<th>Privilege</th>
<th>Mode Dep</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>11100 11100 0111 0100</td>
<td>XX2</td>
<td>632</td>
<td>xcredv</td>
<td>v2.06</td>
<td>VSX Scalar Reciprocal Estimate Double-Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11100 11100 0001 0100</td>
<td>XX2</td>
<td>633</td>
<td>xredp</td>
<td>v2.06</td>
<td>VSX Scalar Reciprocal Estimate Single-Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X X X X</td>
<td>X</td>
<td>634</td>
<td>xregp[x]</td>
<td>v3.0</td>
<td>VSX Scalar Round Quad-Precision to Integral [Exact]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11111 1111 0100 1000</td>
<td>XX2</td>
<td>636</td>
<td>xregsp</td>
<td>v3.0</td>
<td>VSX Scalar Round Quad-Precision to XP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11110 1111 0110 1000</td>
<td>XX2</td>
<td>638</td>
<td>xregsp</td>
<td>v2.07</td>
<td>VSX Scalar Round Double-Precision to Single-Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11110 1111 0110 1000</td>
<td>XX2</td>
<td>638</td>
<td>xregsp</td>
<td>v2.06</td>
<td>VSX Scalar Reciprocal Square Root Estimate Double-Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11110 1111 0110 1000</td>
<td>XX2</td>
<td>640</td>
<td>xregtexp</td>
<td>v2.07</td>
<td>VSX Scalar Reciprocal Square Root Estimate Single-Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11110 1111 0110 1000</td>
<td>XX2</td>
<td>641</td>
<td>xregtexp</td>
<td>v2.06</td>
<td>VSX Scalar Square Root Double-Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X X X X</td>
<td>X</td>
<td>642</td>
<td>xregtexp</td>
<td>v3.0</td>
<td>VSX Scalar Square Root Double-Precision [with round to Odd]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11110 1111 0110 1000</td>
<td>XX2</td>
<td>644</td>
<td>xregtexp</td>
<td>v2.07</td>
<td>VSX Scalar Square Root Single-Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11110 1111 0110 1000</td>
<td>XX2</td>
<td>645</td>
<td>xregtexp</td>
<td>v2.06</td>
<td>VSX Scalar Subtract Double-Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X X X X</td>
<td>X</td>
<td>647</td>
<td>xregtexp</td>
<td>v3.0</td>
<td>VSX Scalar Subtract Quad-Precision [with round to Odd]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11110 1111 0110 1000</td>
<td>XX2</td>
<td>651</td>
<td>xtestdp</td>
<td>v2.06</td>
<td>VSX Scalar Test for software Divide Double-Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11110 1111 0110 1000</td>
<td>XX2</td>
<td>652</td>
<td>xtestdp</td>
<td>v2.06</td>
<td>VSX Scalar Test for software Square Root Double-Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11110 1111 0110 1000</td>
<td>XX2</td>
<td>653</td>
<td>xtestdcep</td>
<td>v3.0</td>
<td>VSX Scalar Test Data Class Double-Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X X X X</td>
<td>X</td>
<td>654</td>
<td>xtestdcep</td>
<td>v3.0</td>
<td>VSX Scalar Test Data Class Double-Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11110 1111 0110 1000</td>
<td>XX2</td>
<td>655</td>
<td>xtestdcep</td>
<td>v3.0</td>
<td>VSX Scalar Test Data Class Single-Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X X X X</td>
<td>X</td>
<td>656</td>
<td>xtestdcep</td>
<td>v3.0</td>
<td>VSX Scalar Extract Exponent Double-Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11110 1111 0110 1000</td>
<td>X</td>
<td>656</td>
<td>xtestdcep</td>
<td>v3.0</td>
<td>VSX Scalar Extract Exponent Double-Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X X X X</td>
<td>X</td>
<td>657</td>
<td>xtestdcep</td>
<td>v3.0</td>
<td>VSX Scalar Extract Exponent Double-Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11110 1111 0110 1000</td>
<td>X</td>
<td>657</td>
<td>xtestdcep</td>
<td>v3.0</td>
<td>VSX Scalar Extract Exponent Double-Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X X X X</td>
<td>X</td>
<td>658</td>
<td>xtestdcep</td>
<td>v2.06</td>
<td>VSX Vector Absolute Double-Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11110 1111 0110 1000</td>
<td>X</td>
<td>658</td>
<td>xtestdcep</td>
<td>v2.06</td>
<td>VSX Vector Absolute Double-Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X X X X</td>
<td>X</td>
<td>659</td>
<td>xtestdcep</td>
<td>v2.06</td>
<td>VSX Vector Absolute Double-Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X X X X</td>
<td>X</td>
<td>663</td>
<td>xtestdcep</td>
<td>v2.06</td>
<td>VSX Vector Absolute Double-Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11110 1111 0110 1000</td>
<td>X</td>
<td>665</td>
<td>xcmpeqdp</td>
<td>v2.06</td>
<td>VSX Vector Compare Equal Double-Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11110 1111 0110 1000</td>
<td>X</td>
<td>666</td>
<td>xcmpeqdp</td>
<td>v2.06</td>
<td>VSX Vector Compare Equal Double-Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X X X X</td>
<td>X</td>
<td>667</td>
<td>xcmpeqdp</td>
<td>v2.06</td>
<td>VSX Vector Compare Greater Than or Equal Double-Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11110 1111 0110 1000</td>
<td>X</td>
<td>667</td>
<td>xcmpeqdp</td>
<td>v2.06</td>
<td>VSX Vector Compare Greater Than or Equal Double-Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X X X X</td>
<td>X</td>
<td>668</td>
<td>xcmpeqdp</td>
<td>v2.06</td>
<td>VSX Vector Compare Greater Than or Equal Double-Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11110 1111 0110 1000</td>
<td>X</td>
<td>669</td>
<td>xcmpeqdp</td>
<td>v2.06</td>
<td>VSX Vector Compare Greater Than or Equal Double-Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11110 1111 0110 1000</td>
<td>X</td>
<td>670</td>
<td>xcmpeqdp</td>
<td>v2.06</td>
<td>VSX Vector Compare Greater Than or Equal Double-Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11110 1111 0110 1000</td>
<td>X</td>
<td>671</td>
<td>xcmpeqdp</td>
<td>v2.06</td>
<td>VSX Vector Compare Greater Than Single-Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X X X X</td>
<td>X</td>
<td>671</td>
<td>xcmpeqdp</td>
<td>v2.06</td>
<td>VSX Vector Compare Greater Than Single-Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X X X X</td>
<td>X</td>
<td>671</td>
<td>xcmpeqdp</td>
<td>v2.06</td>
<td>VSX Vector Compare Greater Than Single-Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11110 1111 0110 1000</td>
<td>X</td>
<td>672</td>
<td>xcmpeqdp</td>
<td>v2.06</td>
<td>VSX Vector Compare Greater Than Single-Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X X X X</td>
<td>X</td>
<td>673</td>
<td>xcmpeqdp</td>
<td>v2.06</td>
<td>VSX Vector Compare Greater Than Single-Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11110 1111 0110 1000</td>
<td>X</td>
<td>675</td>
<td>xcmpeqdp</td>
<td>v2.06</td>
<td>VSX Vector Compare Greater Than Single-Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X X X X</td>
<td>X</td>
<td>677</td>
<td>xcmpeqdp</td>
<td>v2.06</td>
<td>VSX Vector Compare Greater Than Single-Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11110 1111 0110 1000</td>
<td>X</td>
<td>679</td>
<td>xcmpeqdp</td>
<td>v2.06</td>
<td>VSX Vector Compare Greater Than Single-Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11110 1111 0110 1000</td>
<td>X</td>
<td>681</td>
<td>xcmpeqdp</td>
<td>v2.06</td>
<td>VSX Vector Compare Greater Than Single-Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X X X X</td>
<td>X</td>
<td>682</td>
<td>xcmpeqdp</td>
<td>v2.06</td>
<td>VSX Vector Compare Greater Than Single-Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X X X X</td>
<td>X</td>
<td>683</td>
<td>xcmpeqdp</td>
<td>v2.06</td>
<td>VSX Vector Compare Greater Than Single-Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11110 1111 0110 1000</td>
<td>X</td>
<td>684</td>
<td>xcmpeqdp</td>
<td>v2.06</td>
<td>VSX Vector Compare Greater Than Single-Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X X X X</td>
<td>X</td>
<td>686</td>
<td>xcmpeqdp</td>
<td>v2.06</td>
<td>VSX Vector Compare Greater Than Single-Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11110 1111 0110 1000</td>
<td>X</td>
<td>688</td>
<td>xcmpeqdp</td>
<td>v2.06</td>
<td>VSX Vector Compare Greater Than Single-Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X X X X</td>
<td>X</td>
<td>689</td>
<td>xcmpeqdp</td>
<td>v2.06</td>
<td>VSX Vector Compare Greater Than Single-Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11110 1111 0110 1000</td>
<td>X</td>
<td>690</td>
<td>xcmpeqdp</td>
<td>v2.06</td>
<td>VSX Vector Compare Greater Than Signed Doubleword format</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X X X X</td>
<td>X</td>
<td>692</td>
<td>xcmpeqdp</td>
<td>v2.06</td>
<td>VSX Vector Compare Greater Than Signed Doubleword format</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 90. Power ISA AS Instruction Set Sorted by Mnemonic (Sheet 16 of 18)
Appendix F. Power ISA Instruction Set Sorted by Mnemonic  

<table>
<thead>
<tr>
<th>Instruction1</th>
<th>Format</th>
<th>Book</th>
<th>Page</th>
<th>Mnemonic</th>
<th>Version</th>
<th>Privilege3</th>
<th>Mode Dep4</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>111100 ...... ///// ...... 01000 1010..</td>
<td>XX2</td>
<td>692</td>
<td>xvcvxsdsp</td>
<td>v2.06</td>
<td>VSX Vector Convert with round Signed Word to Double-Precision format</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>111100 ...... ///// ...... 01001 1001..</td>
<td>XX2</td>
<td>693</td>
<td>xvcvxswdp</td>
<td>v2.06</td>
<td>VSX Vector Convert Signed Word to Double-Precision format</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>111100 ...... ///// ...... 01010 1001..</td>
<td>XX2</td>
<td>693</td>
<td>xvcvxswsp</td>
<td>v2.06</td>
<td>VSX Vector Convert with round Signed Word to Single-Precision format</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>111100 ...... ///// ...... 01011 0101..</td>
<td>XX2</td>
<td>694</td>
<td>xvcvxvuxdp</td>
<td>v2.06</td>
<td>VSX Vector Convert with round Unsigned Doubleword to Double-Precision format</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>111100 ...... ///// ...... 01010 1011..</td>
<td>XX2</td>
<td>694</td>
<td>xvcvxvuxsp</td>
<td>v2.06</td>
<td>VSX Vector Convert with round Unsigned Doubleword to Single-Precision format</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>111100 ...... ///// ...... 01000 1000..</td>
<td>XX2</td>
<td>695</td>
<td>xvcvxvuxwp</td>
<td>v2.06</td>
<td>VSX Vector Convert with round Unsigned Word to Single-Precision format</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 90. Power ISA AS Instruction Set Sorted by Mnemonic (Sheet 17 of 18)
<table>
<thead>
<tr>
<th>Instruction¹</th>
<th>Format</th>
<th>Book</th>
<th>Page</th>
<th>Mnemonic</th>
<th>Version²</th>
<th>Privilege³</th>
<th>Mode Dep⁴</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>111100 ... 111100 ... 01100 1011</td>
<td>XX2</td>
<td>I</td>
<td>751</td>
<td>xvsqrtsp</td>
<td>v2.06</td>
<td>VSX Vector Square Root Single-Precision</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111100 ... 111100 ... 01000 1011</td>
<td>XX2</td>
<td>I</td>
<td>752</td>
<td>xvsqrtsp</td>
<td>v2.06</td>
<td>VSX Vector Square Root Single-Precision</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111100 ... 111100 ... 00101 1011</td>
<td>XX3</td>
<td>I</td>
<td>753</td>
<td>xvsqrtsp</td>
<td>v2.06</td>
<td>VSX Vector Square Root Double-Precision</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111100 ... 111100 ... 01010 1011</td>
<td>XX3</td>
<td>I</td>
<td>754</td>
<td>xvsqrtsp</td>
<td>v2.06</td>
<td>VSX Vector Square Root Single-Precision</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111100 ... 111100 ... 00100 1011</td>
<td>XX3</td>
<td>I</td>
<td>755</td>
<td>xvsqrtsp</td>
<td>v2.06</td>
<td>VSX Vector Square Root Single-Precision</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111100 ... 111100 ... 01001 1011</td>
<td>XX3</td>
<td>I</td>
<td>756</td>
<td>xvsqrtsp</td>
<td>v2.06</td>
<td>VSX Vector Square Root Single-Precision</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111100 ... 111100 ... 00110 1011</td>
<td>XX3</td>
<td>I</td>
<td>757</td>
<td>xvsqrtsp</td>
<td>v2.06</td>
<td>VSX Vector Square Root Single-Precision</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111100 ... 111100 ... 00011 1011</td>
<td>XX3</td>
<td>I</td>
<td>758</td>
<td>xvsqrtsp</td>
<td>v2.06</td>
<td>VSX Vector Square Root Single-Precision</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111100 ... 111100 ... 00010 1011</td>
<td>XX3</td>
<td>I</td>
<td>759</td>
<td>xvsqrtsp</td>
<td>v2.06</td>
<td>VSX Vector Square Root Single-Precision</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111100 ... 111100 ... 10010 1011</td>
<td>XX3</td>
<td>I</td>
<td>760</td>
<td>xvsqrtsp</td>
<td>v2.06</td>
<td>VSX Vector Square Root Single-Precision</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111100 ... 111100 ... 10001 1011</td>
<td>XX3</td>
<td>I</td>
<td>761</td>
<td>xvsqrtsp</td>
<td>v2.06</td>
<td>VSX Vector Square Root Single-Precision</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111100 ... 111100 ... 10000 1011</td>
<td>XX3</td>
<td>I</td>
<td>762</td>
<td>xvsqrtsp</td>
<td>v2.06</td>
<td>VSX Vector Square Root Single-Precision</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111100 ... 111100 ... 10111 1011</td>
<td>XX3</td>
<td>I</td>
<td>763</td>
<td>xvsqrtsp</td>
<td>v2.06</td>
<td>VSX Vector Square Root Single-Precision</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111100 ... 111100 ... 10110 1011</td>
<td>XX3</td>
<td>I</td>
<td>764</td>
<td>xvsqrtsp</td>
<td>v2.06</td>
<td>VSX Vector Square Root Single-Precision</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111100 ... 111100 ... 10101 1011</td>
<td>XX3</td>
<td>I</td>
<td>765</td>
<td>xvsqrtsp</td>
<td>v2.06</td>
<td>VSX Vector Square Root Single-Precision</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111100 ... 111100 ... 10100 1011</td>
<td>XX3</td>
<td>I</td>
<td>766</td>
<td>xvsqrtsp</td>
<td>v2.06</td>
<td>VSX Vector Square Root Single-Precision</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111100 ... 111100 ... 10011 1011</td>
<td>XX3</td>
<td>I</td>
<td>767</td>
<td>xvsqrtsp</td>
<td>v2.06</td>
<td>VSX Vector Square Root Single-Precision</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111100 ... 111100 ... 10010 1011</td>
<td>XX3</td>
<td>I</td>
<td>768</td>
<td>xvsqrtsp</td>
<td>v2.06</td>
<td>VSX Vector Square Root Single-Precision</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111100 ... 111100 ... 10000 1011</td>
<td>XX3</td>
<td>I</td>
<td>769</td>
<td>xvsqrtsp</td>
<td>v2.06</td>
<td>VSX Vector Square Root Single-Precision</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 90. Power ISA AS Instruction Set Sorted by Mnemonic (Sheet 18 of 18)

1. Key to Instruction column.
   /
   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 having a value 0.
   1
   Instruction bit having a value 1.

2. Key to Version column.

- P1 Instruction introduced in the POWER Architecture.
- P2 Instruction introduced in the POWER2 Architecture.
- PPC Instruction introduced in the PowerPC Architecture prior to v2.00.
- v2.00 Instruction introduced in the PowerPC Architecture Version 2.00.
- v2.01 Instruction introduced in the PowerPC Architecture Version 2.01.
- v2.02 Instruction introduced in the PowerPC Architecture Version 2.02.
- v2.03 Instruction introduced in the Power ISA Architecture Version 2.03.
- v2.04 Instruction introduced in the Power ISA Architecture Version 2.04.
- v2.05 Instruction introduced in the Power ISA Architecture Version 2.05.
- v2.06 Instruction introduced in the Power ISA Architecture Version 2.06.
- v2.07 Instruction introduced in the Power ISA Architecture Version 2.07.
- v3.0 Instruction introduced in the Power ISA Architecture Version 3.0.
- v3.0B Instruction introduced in the Power ISA Architecture Version 3.0B.
3. Key to Privilege column.

- **P**: Denotes an instruction that is treated as privileged.
- **O**: Denotes an instruction that is treated as privileged or nonprivileged (or hypervisor, for mtspr), depending on the SPR or PMR number.
- **PI**: Denotes an instruction that is illegal in privileged state.
- **H**: Denotes an instruction that can be executed only in hypervisor state.
- **U**: Denotes an instruction that can be executed only in ultravisor state.

4. Key to Mode Dependency column.

Except as described below and in Section 1.11.3, “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.
- **32**: The instruction can be executed only in 32-bit mode.
- **64**: The instruction can be executed only in 64-bit mode.
Last Page - End of Document