OpenPOWER Foundation
Power Instruction Set Architecture

Instruction Set Architecture Specification Document

Power ISA™
Version 3.1C
May 26, 2024

revision : 99a1cac - date : 2023-06-22

Power ISA™

Copyright © OpenPOWER Foundation 2021.
All Rights Reserved.

Copyright © IBM Corporation 1994, 2021.

By downloading or using the POWER® Instruction Set Architecture (“ISA”) Specification, you agree to be bound by the terms and conditions of the OpenPOWER Power ISA End User License Agreement (EULA).

OpenPOWER, the OpenPOWER logo, and openpowerfoundation.org are trademarks or registered trademarks of OpenPOWER Foundation, Inc., registered in many jurisdictions worldwide.

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. A current list of IBM trademarks is available on the Web at “Copyright and trademark information” at www.ibm.com/legal/copytrade.shtml.

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

IBM®
Power ISA
PowerPC®
Power Architecture®
PowerPC Architecture
RISC System/6000
POWER®
POWER2
POWER4
POWER5
POWER7®
POWER8®
POWER9®
Power10™
System/370

Other company, product, and service names may be trademarks or service marks of others.

OpenPOWER Power ISA End User License Agreement

"*Power ISA*" means the microprocessor instruction set architecture specification version provided to you with this document. By exercising any right under this End User License Agreement, you ("Recipient") agree to be bound by the terms and conditions of this Power ISA End User License ("Agreement").

All information contained in the Power ISA is subject to change without notice. The products described in the Power ISA are NOT intended for use in applications such as implantation, life support, or other hazardous uses where malfunction could result in death, bodily injury, or catastrophic property damage.

Definitions

"*Architectural Resources*" means assignable resources necessary for elements of the Power ISA to interoperate, including, but not limited to: opcodes, special purpose registers, defined registers, reserved bits in existing defined registers, control table fields and bits, and interrupt vectors.

"*Compliancy Subset*" means a portion of the Power ISA, defined within the Power ISA, which must be implemented to ensure software compatibility across Power ISA compliant devices.

"*Contribution*" means any work of authorship that is intentionally submitted to OPF for inclusion in the Power ISA by the copyright owner or by an individual or entity authorized to submit on behalf of the copyright owner. Without limiting the generality of the preceding sentence, RFCs will be considered Contributions.

"*Custom Extensions*" means additions to the Power ISA in a designated subset of Architectural Resources defined by the Power ISA. For clarity, Custom Extensions are not Contributions.

"*Integrated Circuit*" shall mean an integral unit including a plurality of active and passive circuit elements formed at least in part of semiconductor material arranged on or in a chip(s) or substrate.

"*OPF*" means The OpenPOWER Foundation.

"*Licensed Patent Claims*" means patent claims:

(a) licensable by or through OPF; and

(b) which, but for this Agreement, would be necessarily infringed by the use of the Power ISA in making, using, or otherwise implementing a Power Compliant Chip.

"*Party(ies)*" means Recipient or OPF or both.

"*OpenPOWER ISA Compliance Definition*" means the validation procedures associated with architectural compliance developed, delivered, and maintained by and available from OPF.

"*Power Compliant*" means an implementation of (i) one of the Compliancy Subsets alone or (ii) one of the Compliancy Subsets together with selected permitted combinations of additional instructions and/or facilities within the Power ISA, in the case of clauses (i) and (ii), provided that such implementation meets the corresponding portions of the OpenPOWER ISA Compliance Definition.

"*Power ISA Core*" means an implementation of the Power ISA that is represented by software, a hardware description language (HDL), or an Integrated Circuit design, but excluding physically implemented chips (such as microprocessors, system on a chips, or field-programmable gate arrays (FPGAs)); provided that such implementation is primarily designed to be included as part of software, a hardware description language (HDL), or an Integrated Circuit design that are in each case Power Compliant, regardless of whether such implementation, independently, is Power Compliant.

"*Power Compliant Chip*" means a Power Compliant physical implementation of one or more Power ISA Cores into one or more Integrated Circuits, including, for example, in a microprocessor, system on a chip, or a field-programmable gate array (FPGA), provided that all portions of such physical implementation are Power Compliant.

"*Request for Change (RFC)*" means any request for change in the Power ISA as a whole or a change in the definition of a Compliancy Subset provided in the Power ISA.

1. Grant of Rights

Solely for the purposes of developing and expanding the Power ISA and the POWER ecosystem, and subject to the terms of this Agreement:

1.1 OPF grants to Recipient a nonexclusive, worldwide, perpetual, royalty-free, non-transferable license under all copyrights licensable by OPF and contained in the Power ISA to a) develop technology products compatible with the Power ISA, and b) create, use, reproduce, perform, display, and distribute Power ISA Cores.

1.2 OPF grants to Recipient the right to license Recipient Power ISA Cores under the Creative Commons Attribution 4.0 license.

1.3 OPF grants to Recipient the right to sell or license Recipient Power ISA Cores under independent terms that are consistent with the rights and licenses granted under this Agreement. As a condition of the license grant under this section 1.3, the Recipient must either provide the Power ISA with this Agreement to the downstream recipient, or provide notification for the downstream recipient to obtain the Power ISA and this Agreement to have appropriate patent licenses to implement the Power ISA Core as a Power Compliant Chip. It is clarified that no rights are to be granted under this Section 1.3 beyond what is expressly permitted by this Agreement.

1.4 Notwithstanding Sections 1.1 through 1.3 above, Recipient shall not have the right or license to create, use, reproduce, perform, display, distribute, sell, or license the Power ISA Core in a physically implemented chip (including a microprocessor, system on a chip, or a field-programmable gate array [FPGA]) that is not Power Compliant, nor to license others to do so.

1.5 OPF grants to Recipient a nonexclusive, worldwide, perpetual, royalty-free, non-transferable license under Licensed Patent Claims to make, use, import, export, sell, offer for sale, and distribute Power Compliant Chips.

1.6 If Recipient institutes patent litigation or an administrative proceeding (including a cross-claim or counterclaim in a lawsuit, or a United States International Trade Commission proceeding) against OPF, OPF members, or any third-party entity (including but not limited to any third party that made a Contribution) alleging infringement of any Recipient patent by any version of the Power ISA, or the implementation thereof in a CPU design, IP core, or chip, then all rights, covenants, and licenses granted by OPF to Recipient under this Agreement shall terminate as of the date such litigation or proceeding is initiated.

1.7 Without limiting any other rights or remedies of OPF, if Recipient materially breaches the terms of this Agreement, OPF may terminate this Agreement at its discretion.

2. Modifications to the Power ISA

2.1 Recipient shall have the right to submit Contributions to the Power ISA through a prospectively authorized process by OPF but shall not implement such Contributions until fully approved through the prospectively authorized OPF process.

2.2 Recipient may create Custom Extensions as described and permitted in the Power ISA. Recipient is encouraged, but not required, to bring their Custom Extensions through the authorized OPF process for contributions. For clarity, Custom Extensions cannot be guaranteed to be compatible with another third-party's Custom Extensions.

3. Ownership

3.1 Nothing in this Agreement shall be deemed to transfer to Recipient any ownership interest in any intellectual property of OPF or of any contributor to the Power ISA, including but not limited to any copyrights, trade secrets, know-how, trademarks associated with the Power ISA or any patents, registrations or applications for protection of such intellectual property.

3.2 Recipient retains ownership of all incremental work done by Recipient to create Power ISA Cores and Power Compliant Chips, subject to the ownership rights of OPF and any contributors to the Power ISA. Nothing in this Agreement shall be deemed to transfer to OPF any ownership interest in any intellectual property of Recipient, including but not limited to any copyrights, trade secrets, know-how, trademarks, patents, registrations, or applications for protection of such intellectual property.

4. Limitation of Liability

4.1 THE POWER ISA AND ANY OTHER INFORMATION CONTAINED IN OR PROVIDED UNDER THIS DOCUMENT ARE PROVIDED ON AN "AS IS" BASIS. OPF makes no representations or warranties, either express or implied, including but not limited to, warranties of merchantability, fitness for a particular purpose, or non-infringement, or that any practice or implementation of the Power ISA or other OPF documentation will not infringe any third-party patents, copyrights, trade secrets, or other rights. In no event will OPF or any other person or entity submitting any Contribution to OPF be liable for

damages arising directly or indirectly from any use of the Power ISA or any other information contained in or provided under this document.

5. Compliance with Law

5.1 Recipient shall be responsible for compliance with all applicable laws, regulations and ordinances, and will obtain all necessary permits and authorizations applicable to the future conduct of its business involving the Power ISA. Recipient agrees to comply with all applicable international trade laws and regulations such as export controls, embargo/sanctions, antiboycott, and customs laws related to the future conduct of the business involving the Power ISA to be transferred under this Agreement. Recipient warrants that it is knowledgeable with, and will remain in full compliance with, all applicable export controls and embargo/sanctions laws, regulations or rules, orders, and policies, including but not limited to, the U.S. International Traffic in Arms Regulations ("ITAR"), the U.S. Export Administration Regulations ("EAR"), and the regulations of the Office of Foreign Assets Control ("OFAC"), U.S. Department of Treasury.

6. Choice of Law

6.1 This Agreement is governed by the laws of the State of New York, without regard to the conflict of law provisions thereof.

7. Publicity

7.1 Nothing contained in these terms shall be construed as conferring any right to use in advertising, publicity or other promotional activities any name, trade name, trademark or other designation of any Party hereto (including any contraction, abbreviation or simulation of any of the foregoing).

All capitalized terms in the following text have the meanings assigned to them in the OpenPOWER Intellectual Property Rights Policy (the "OpenPOWER IPR Policy"). The full Policy may be found at the OpenPOWER website.

OpenPOWER requests that any OpenPOWER Party or any other party that believes it has patent claims that would necessarily be infringed by implementations of this OpenPOWER Standards Final Deliverable, to notify OpenPOWER WG Administrator and provide an indication of its willingness to grant patent licenses to such patent claims in a manner consistent with the IPR Mode of the OpenPOWER Work Group that produced this deliverable.

OpenPOWER invites any party to contact the OpenPOWER WG Administrator if it is aware of a claim of ownership of any patent claims that would necessarily be infringed by implementations of this OpenPOWER Standards Final Deliverable by a patent holder that is not willing to provide a license to such patent claims in a manner consistent with the IPR Mode of the OpenPOWER Work Group that produced this OpenPOWER Standards Final Deliverable. OpenPOWER may include such claims on its website but disclaims any obligation to do so.

OpenPOWER takes no position regarding the validity or scope of any intellectual property or other rights that might be claimed to pertain to the implementation or use of the technology described in this OpenPOWER Standards Final Deliverable or the extent to which any license under such rights might or might not be available; neither does it represent that it has made any effort to identify any such rights. Information on OpenPOWER procedures with respect to rights in any document or deliverable produced by an OpenPOWER Work Group can be found on the OpenPOWER website. Copies of claims of rights made available for publication and any assurances of licenses to be made available, or the result of an attempt made to obtain a general license or permission for the use of such proprietary rights by implementers or users of this OpenPOWER Standards Final Deliverable, can be obtained from the OpenPOWER WG Administrator. OpenPOWER makes no representation that any information or list of intellectual property rights will at any time be complete, or that any claims in such list are, in fact, Essential Claims.

Preface

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

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

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

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

The Power ISA Version 3.1C 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.

In this document, changes from the Power ISA Version 3.0C are indicated in green. Places where significant text has been deleted are indicated with a "[]" symbol.

Summary of Changes in Power ISA Version 3.1C

This document is Version 3.1C of the Power ISA. It is intended to supersede and replace version 3.0C. Any product descriptions that reference a version of the architecture are understood to reference the latest version. This version was created by making miscellaneous corrections and by applying the following requests for change (RFCs) to Power ISA Version 3.0C. In contrast to change indications elsewhere in the document, change indications in this summary of changes indicate changes relative to v3.1B.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

VSX PCV Generate Operations:
Added new permute control vector generate instructions to support efficient emulation of load expand and store compress operations.

New Performance Monitor SPRs:
Added three new performance monitor SPRs. SIER2 and SIER3 are added to provide additional information about the sampled instruction. MMCR3 is added for further sampling related configuration control.

Translation Management Extensions:
Added an L bit for slbiag, where L=1 indicates an invalidation by LPID. tlbiel with SET=0 and IS=1, 2, or 3 invalidate all congruence classes and tlbiel with SET !=0

is a noop except when RIC=1, which becomes an invalid form. Made ISL apply in hypervisor state.

Copy/Paste Extensions:
Added memory move functionality.

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

Dynamic Execution Control Facility:
Added support for dynamic software control of security vs performance aspect during execution of a hardware thread.

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

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

Hash-based Pointer Authentication:
Added support for hash-based pointer authentication scheme. Primary application of this scheme is protection against return-oriented programming (ROP).

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

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

BHRB Disable Control:
Adds an additional control on BHRB recording via MMCRA bit 26 namely $MMCRA_{BHRBRD}$.

Translation Management Extensions #2:
Added SW bit to the radix PDE. Added the term "effR" in an attempt to clarify tlbie[l] function. Allowed for the TLB entry not to have the base (or actual) page size(s).

Exception related fixes for MMA:
When underflow is enabled, simple underflow (when the result is not also inexact) causes an exception. Exception behavior for bfloat16 converts is made similar to other converts instead of similar to the GERs. Underflow and overflow trap enabled exception bit setting is corrected. The name and mnemonic for xvcvbf16sp are changed to indicate it generates no floating-point exceptions.

Storage Interrupt Cleanup:
Cleaned up the presentations of [H]DSI and [H]ISI and makes related changes elsewhere. Also removed application access to Radix Quadrant 3.

Performance Monitor Threshold Counter Multiplier Width:
Extended the Threshold Event Counter Multiplier field (TECM) specified by MMCRA SPR from 7 bits [38:44] to 8 bits [37:44]. Added the interpretation of Sampled Instruction Type SITYPE=0b111 encoding given by SIER[46:48].

VSX Load/Store with Length Atomicity:
Defined circumstances under which the accesses caused by the VSX load/store with length instructions are single copy atomic. Removed the special case exclusion for single copy atomicity of accesses to the individual words or doublewords of load/store multiple. Applied Java decomposition rules to the VSX load/store with length instructions and the individual word/doubleword accesses of load/store multiple.

Synchronization Clarifications:
Made various clarifications related to the synchronization of translation table updates and invalidations.

OpenISA Compliancy Subset Methodology and Requirements

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

OpenPOWER Compliancy Subsets

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

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

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

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

OpenPOWER Optional and Deprecated Features

Always Optional Features

The following features are optional for all compliancy subsets.

Feature	Reference
Copy/Paste for accelerator invocation and memory copy (CPA)	See Section 4.4 of Book II.
Secure Memory Facility (SMF)[1]	See Chapter 3 of Book III.
Hardware and software data stream prefetching (STM) (DSCR state not optional)	See Section 4.2 and Section 4.3.2 of Book II.
M=0 (M) (non-coherent memory)	See Section 1.6.3 of Book II.
W=1 (W) (write through-required memory)	See Section 1.6.1 of Book II.
Power management (PM)[2]	See Section 4.2.2, Section 4.3.2, and the description of the PECE field(s) of the LPCR in Section 2.2 of Book III.
MMA[3]	See Section 7.2.1.3 and Section 7.6.2.12 of Book I.

Notes:

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

Linux Optional Features

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

Feature	Reference
AIL/HAIL programmability (AIL) (AIL=3 and HAIL=1 required)	See the description of the AIL and HAIL fields of the LPCR in Section 2.2 of Book III.
Atomic Memory Operations (AMO)	See Section 4.5 of Book II.
Big Endian (BE) (LE is required for LCS. Linux supporting LCS is 64b LE Linux.)	See Section 1.10 of Book I and its first two subsections. Also see the description of the ILE field of the LPCR in Section 2.2 of Book III and the description of the LE bit of the MSR in Section 4.2.1 of Book III.
Branch History Rolling Buffer (BHRB)	See Chapter 7 of Book II.
Decimal floating-point (DFP)[1]	See Chapter 5 of Book I.
Event-Based Branching (EBB)	See Chapter 6 of Book II.
EVIRT programmability (EVIRT)[2] (EVIRT=1 required)	See the description of the EVIRT field of the LPCR in Section 2.2 of Book III.
SLB / HPT translation (HPT) (includes VPM, ISL, KBV)	See Section 6.7.7 through Section 6.7.9 of Book III. Also see the description of the VPM, ISL, and KBV fields of the LPCR in Section 2.2 of Book III.
Load/Store Multiple instructions (LM)	See Section 3.3.6 of Book I.
Load/Store String instructions (LS)	See Section 3.3.7 of Book I.
Processor Compatibility Register (PCR)[2]	See Section 2.5 of Book III.
Quad-precision floating-point (QFP)[3]	See Chapter 7 of Book I.
Broadcast TLB shootdown (TLBIE) (tlbiel not optional)	See Section 6.9.3.3 of Book III.
Control Register (CTRL)	See Section 5.3.4 of Book III.
SMT (SMT)[4] (includes PURR/SPURR, PSPB, RPR, PPR, processor control) (PPR and hypervisor/ultravisor messaging not optional)	See Chapter 3 of Book II. Also see Section 5.3.5 through Section 5.3.7, Section 8.6, Section 8.7, and Chapter 12 of Book III.

Notes:

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

Scalar Float Optional Features

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

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

Notes:

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

Scalar Fixed Optional Features

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

Feature	Reference
Scalar binary floating-point (FP)	See Chapter 4 of Book I.

Deprecated Features

There are no deprecated features in Power ISA v3.1C.

OpenPOWER Architecture Sandbox

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

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

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

The architecture sandbox consists of the following.

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

Table of Contents

List of Tables

I. Power ISA User Instruction Set Architecture

II. Power ISA Virtual Environment Architecture

III. Power ISA Operating Environment Architecture

Power ISA Book I-III Appendices

List of Figures

I. Power ISA User Instruction Set Architecture

II. Power ISA Virtual Environment Architecture

III. Power ISA Operating Environment Architecture

Book I

Power ISA User Instruction Set Architecture

Chapter 1. Introduction

1.1 Overview

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

1.2 Instruction Mnemonics and Operands

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

```
stw     RS,D(RA)
addis   RT,RA,SI
```

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

1.3 Document Conventions

1.3.1 Definitions

The following definitions are used throughout this document.

- **program**
  A sequence of related instructions.
- **application program**
  A program that uses only the instructions and resources described in Books I and II.
- **processor**
  The hardware component that implements the instruction set, storage model, and other facilities defined in the Power ISA architecture, and executes the instructions specified in a program.
- **octword, quadword, doubleword, word, halfword, byte, and nibble**
  256 bits, 128 bits, 64 bits, 32 bits, 16 bits, 8 bits, and 4 bits, respectively.
- **positive**
  Means greater than zero.
- **negative**
  Means less than zero.
- **floating-point single format** (or simply **single format**)
  Refers to the representation of a single-precision binary floating-point value in a register or storage.
- **floating-point double format** (or simply **double format**)
  Refers to the representation of a double-precision binary floating-point value in a register or storage.
- **system library program**
  A component of the system software that can be called by an application program using a *Branch* instruction.
- **system service program**
  A component of the system software that can be called by an application program using a *System Call* or *System Call Vectored* instruction.
- **system trap handler**
  A component of the system software that receives control when the conditions specified in a *Trap* instruction are satisfied.
- **system error handler**
  A component of the system software that receives control when an error occurs. The system error handler includes a component for each of the various kinds of error. These error-specific components are referred to as the system alignment error handler, the system data storage error handler, etc.
- **latency**
  Refers to the interval from the time an instruction begins execution until it produces a result that is available for use by a subsequent instruction.
- **unavailable**
  Refers to a resource that cannot be used by the program. For example, storage is unavailable if access to it is denied. See Book III.

- **undefined value**
  May vary between implementations, and between different executions on the same implementation, and similarly for register contents, storage contents, etc., that are specified as being undefined.
- **boundedly undefined**
  The results of executing a given instruction are said to be boundedly undefined if they could have been achieved by executing an arbitrary finite sequence of instructions (none of which yields boundedly undefined results) in the state the processor was in before executing the given instruction. Boundedly undefined results may include the presentation of inconsistent state to the system error handler as described in Section 1.8.1 of Book II. Boundedly undefined results for a given instruction may vary between implementations, and between different executions on the same implementation.
- **"must"**
  If software violates a rule that is stated using the word "must" (e.g., "this field must be set to 0"), the results are boundedly undefined unless otherwise stated.
- **sequential execution model**
  The model of program execution described in Section 2.2, "Instruction Execution Order" on page 33.
- **authority**
  Refers to permission to perform a particular operation.

1.3.2 Notation

The following notation is used throughout the Power ISA documents.

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

  Underscores may be used between digits.
- RT, RA, R1, ... refer to General Purpose Registers.
- FRT, FRA, FR1, ... refer to Floating-Point Registers.
- FRTp, FRAp, FRBp, ... refer to an even-odd pair of Floating-Point Registers. Values must be even, otherwise the instruction form is invalid.
- VRT, VRA, VR1, ... refer to Vector Registers.
- (x) means the contents of register x, where x is the name of an instruction field. For example, (RA) means the contents of register RA, and (FRA) means the contents of register FRA, where RA and FRA are instruction fields. Names such as LR and CTR denote registers, not fields, so parentheses are not used with them. Parentheses are also omitted when register x is the register into which the result of an operation is placed.
- (RA|0) means the contents of register RA if the RA field has the value 1–31, or the value 0 if the RA field is 0.
- Bytes in registers, instructions, fields, and bit strings are numbered from left to right, starting with byte 0 (most significant).
- Bits in registers, instructions, fields, and bit strings are specified as follows. In the last three items (definition of X_p etc.), if X is a field that specifies a GPR, FPR, or VR (e.g., the RS field of an instruction), the definitions apply to the register, not to the field.
  - Bits in instructions, fields, and bit strings are numbered from left to right, starting with bit 0
  - For all registers except the Vector registers, bits in registers that are less than 64 bits start with bit number $64-L$, where L is the register length; for the Vector registers, bits in registers that are less than 128 bits start with bit number $128-L$.
  - The leftmost bit of a sequence of bits is the most significant bit of the sequence.
  - X_p means bit p of register/instruction/field/bit_string X.
  - $X_{p:q}$ means bits p through q of register/instruction/field/bit_string X.
  - $X_{p\,q\,...}$ means bits p, q, ... of register/instruction/field/bit_string X.
- $\neg$(RA) means the one's complement of the contents of register RA.
- A period (.) as the last character of an instruction mnemonic means that the instruction records status information in certain fields of the Condition Register as a side effect of execution.
- The symbol || is used to describe the concatenation of two values. For example, 010 || 111 is the same as 010111.
- x^n means x raised to the n^{th} power.
- ^{n}x means the replication of x, n times (i.e., x concatenated to itself $n-1$ times). ^{n}0 and ^{n}1 are special cases:
  - ^{n}0 means a field of n bits with each bit equal to 0. Thus ^{5}0 is equivalent to 0b00000.
  - ^{n}1 means a field of n bits with each bit equal to 1. Thus ^{5}1 is equivalent to 0b11111.
- Each bit and field in instructions, and in status and control registers (e.g., XER, FPSCR) and Special 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.
- /, //, ///, ... denotes a reserved field, in a register, instruction, field, or bit string.
- ?, ??, ???, ... denotes an implementation-dependent field in a register, instruction, field or bit string.

1.3.3 Reserved Fields, Reserved Values, and Reserved SPRs

Reserved fields in instructions are ignored by the processor.

In some cases a defined field of an instruction has certain values that are reserved. This includes cases in which the field is shown in the instruction layout as containing a particular value; in such cases all other values of the field are reserved. In general, if an instruction is coded such that a defined field contains a reserved value the instruction form is invalid; see Section 1.8.2 on page 25. The only exception to the preceding rule is that it does not apply to Reserved and Illegal classes of instructions (see Section 1.6.2) 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. ***mtspr*** and ***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.

Architecture Note

The requirement that reserved fields in instructions be ignored by the processor, coupled with the requirement that software code these fields as zero, permits the fields to be used in the future to define new instruction variants in a manner that provides both backward and forward compatibility for software. For backward compatibility, the semantics of the new variant must be such that it is architecturally permissible to implement the variant as if it were the base instruction (i.e., to ignore the contents of the newly used field). For forward compatibility, the value zero in the newly used field must cause the instruction to obey the “old” semantics (i.e., the semantics the instruction had before the new variant was added).

Decisions regarding assignment of a meaning for a given reserved field in a given instruction must include consideration of the extent to which the field is ignored by processors that comply with versions of the architecture that precede Version 2.01 and, for such processors that do not ignore the field, of the effect of executing the instruction with nonzero values in the field.

Related considerations apply to reserved bits in System Registers.

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.

Engineering Note

Reserved bits in System Registers need not be implemented, provided that reading any such bits that are not implemented always returns 0.

1.3.4 Description of Instruction Operation

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

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

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

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

Notation	Meaning
$\leftarrow$	Assignment
$\leftarrow_{iea}$	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.
$\neg$	NOT logical operator
$+$	Two's complement addition
$-$	Two's complement subtraction, unary minus
$\times$	Multiplication
$\times_{si}$	Signed-integer multiplication
$\times_{ui}$	Unsigned-integer multiplication
$/$	Division
$\div$	Division, with result truncated to integer
%	Remainder of integer division
$\sqrt{}$	Square root
$=, \neq$	Equals, Not Equals relations
$<, \leq, >, \geq$	Signed comparison relations
$<^u, >^u$	Unsigned comparison relations
?	Unordered comparison relation
&, |	AND, OR logical operators
$\oplus, \equiv$	Exclusive OR, Equivalence logical operators $((a \equiv b) = (a \oplus \neg b))$

ABS(x) Absolute value of x

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

CEIL(x) Least integer $\geq$ x

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

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

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

FLOOR(x) Greatest integer $\leq$ x

GPR(x) General Purpose Register x

LENGTH(x) Length of x, in bits.

MASK(x, y) 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.

$MEM_{metadata}$(x,y) Metadata associated with MEM(x,y).

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

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

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

SPR(x) Special Purpose Register x

TRAP Invoke the system trap handler

x.bit[y] Return the contents of bit y of x.

x.nibble[y] Return the contents of 4-bit nibble element y of x.

x.byte[y] Return the contents of 8-bit byte element y of x.

x.hword[y] Return the contents of 16-bit halfword element y of x.

x.word[y] Return the contents of 32-bit word element y of x.

x.dword[y] Return the contents of 64-bit doubleword element y of x.

x.qword[y] Return the contents of 128-bit quadword element y of x.

x <<< y Result of rotating x left by y bits.
b ← LENGTH(x)
result ← $x_{y:b-1}$ || $x_{0:y-1}$

x >>> y Result of rotating x right by y bits.
b ← LENGTH(x)
result ← $x_{b-y:b-1}$ || $x_{0:b-1-y}$

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 1.1. 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.

Operator precedence	
Operators	**Associativity**
subscript, function evaluation	left to right
pre-superscript (replication), post-superscript (exponentiation)	*right to left*
unary $-$, $\neg$	*right to left*
$\times$, $\div$	left to right
$+$, $-$,	left to right
||	left to right
$=$, $\neq$, $<$, $\leq$, $>$, $\geq$, $<^u$, $>^u$, ?	left to right
&, $\oplus$, $\equiv$	left to right
|	left to right
: (range)	none
$\leftarrow$, $\leftarrow_{iea}$	none

Table 1.1: Operator precedence

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

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

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

Signed integers are represented in two's complement form.

There are no computational instructions that modify storage; instructions that reference storage may reformat the data (e.g. load halfword algebraic). To use a storage operand in a computation and then modify the same or another storage location, the contents of the storage operand must be loaded into a register, modified, and then stored back to the target location. Figure 1.1 is a logical representation of instruction processing. Figure 1.2 shows the registers that are defined in Book I. (A few additional registers that are available to application programs are defined in other Books, and are not shown in the figure.)

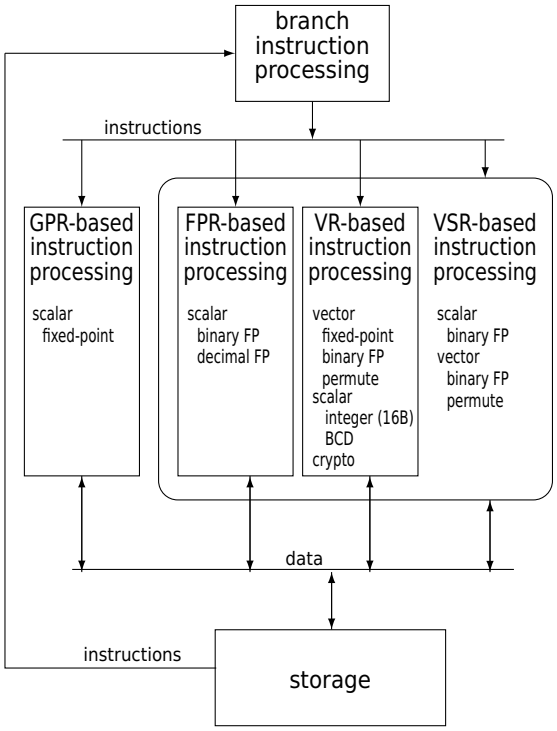


Figure 1.1: Logical processing model

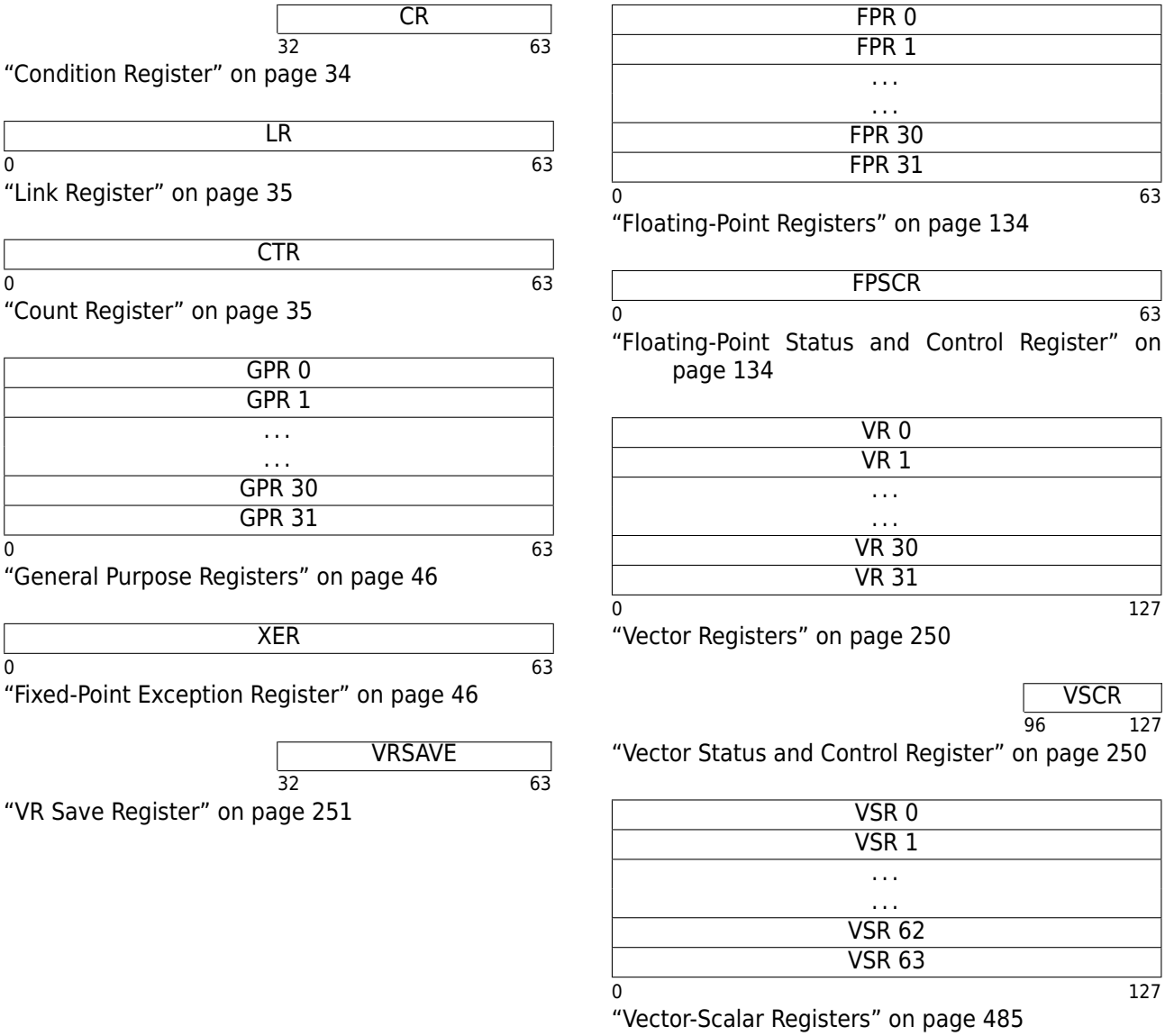


Figure 1.2: Registers that are defined in Book I

1.5 Computation modes

Processors provide two execution modes, 64-bit mode and 32-bit mode. In both of these modes, instructions that set a 64-bit register affect all 64 bits. The computational mode controls how the effective address is interpreted, how Condition Register bits and XER bits are set, how the Link Register is set by *Branch* instructions in which LK=1, and how the Count Register is tested by *Branch Conditional* instructions. Nearly all instructions are available in both modes (the only exceptions are a few instructions that are defined in Book III). In both modes, effective address computations use all 64 bits of the relevant registers (General Purpose Registers, 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 1.10.3 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

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

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

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

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

Architecture Note

For ease of implementation, values corresponding to (i.e., having the same 32-bit binary value) the following word instructions must not appear as the suffix of a prefixed instruction:

- all *Branch* instructions,
- the ***rfebb*** instruction,
- all context synchronizing instructions except ***isync***,
- a "Service Processor Attention" instruction (see Appendix C, "Reserved Instructions').

A prefixed instruction that violates this rule is an illegal instruction (or an invalid form of the instruction, in the case of ***pnop***).

Values corresponding to a *Branch* instruction must not appear as the suffix of a prefixed instruction because the value may be interpreted as a *Branch* instruction before it has been determined to be the suffix of a prefixed instruction, possibly causing branch prediction logic to start prefetching based on the predicted branch path.

Values corresponding to an ***rfebb*** instruction, and to context synchronizing instructions other than ***isync***, must not appear as the suffix of a prefixed instruction because the value may be interpreted as one of these instructions before it has been determined to be the suffix of a prefixed instruction, possibly causing instruction fetching to stop. (***isync*** does not have this problem because whether a given instance of ***isync*** needs to stop instruction fetching is determined when the instruction is about to be executed as a word instruction – which will be never if the ***isync*** appears as the suffix of a prefixed instruction.)

Values corresponding to a "Service Processor Attention" instruction must not appear as the suffix of a prefixed instruction because the value may be interpreted as a "Service Processor Attention" instruction before it has been determined to be the suffix of a prefixed instruction, possibly causing instruction fetching to stop.

The Hypervisor Emulation Assistance interrupt that occurs when attempt is made to execute the illegal instruction resolves these problems naturally, in hardware.

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

Programming Note

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

The format diagrams given below show horizontally all valid combinations of instruction fields. See Section 1.6.2, "Instruction Prefix Formats" for definitions of instruction fields defined in the prefix.

Split Field Notation

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

1.6.1 Word Instruction Formats

1.6.1.1 A-FORM

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

Figure 1.3: A instruction format

1.6.1.2 B-FORM

0	6	11	16	30	31
PO	BO	BI	BD	AA	LK

Figure 1.4: B instruction format

1.6.1.3 D-FORM

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

Figure 1.5: D instruction format

1.6.1.4 DQ-FORM

0	6	11	16	28	29	31
PO	RTp	RA	DQ	///		
PO	S	RA	DQ	SX	XO	
PO	T	RA	DQ	TX	XO	

Figure 1.6: DQ instruction format

1.6.1.5 DS-FORM

0	6	11	16	30	31
PO	FRSp	RA	DS	XO	
PO	FRTp	RA	DS	XO	
PO	RS	RA	DS	XO	
PO	RSp	RA	DS	XO	
PO	RT	RA	DS	XO	
PO	VRS	RA	DS	XO	
PO	VRT	RA	DS	XO	

Figure 1.7: DS instruction format

1.6.1.6 DX-FORM

0	6	11	16	26	31
PO	RT	d1	d0	XO	d2
PO	RT	b1	b0	XO	b2

Figure 1.8: DX instruction format

1.6.1.7 I-FORM

0	6	30	31
PO	LI	AA	LK

Figure 1.9: I instruction format

1.6.1.8 M-FORM

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

Figure 1.10: M instruction format

1.6.1.9 MD-FORM

0	6	11	16	21	27	30	31
PO	RS	RA	sh	mb	XO	sh	Rc
PO	RS	RA	sh	me	XO	sh	Rc

Figure 1.11: MD instruction format

1.6.1.10 MDS-FORM

0	6	11	16	21	25	27	31
PO	RS	RA	RB	mb		XO	Rc
PO	RS	RA	RB	me		XO	Rc

Figure 1.12: MDS instruction format

1.6.1.11 SC-FORM

0	6	11	16	20	27	30	31
PO	///	///	///	LEV	///	0	1
PO	///	///	///	LEV	///	1	/

Figure 1.13: SC instruction format

1.6.1.12 VA-FORM

0	6	11	16	21 22	26	31
PO	RT	RA	RB	RC	XO	
PO	VRT	VRA	VRB	/ SHB	XO	
PO	VRT	VRA	VRB	VRC	XO	

Figure 1.14: VA instruction format

1.6.1.13 VC-FORM

0	6	11	16	21 22	31
PO	VRT	VRA	VRB	Rc	XO

Figure 1.15: VC instruction format

1.6.1.14 VX-FORM

0	6	11 12 13 14 15 16	21 22 23	31
PO	///	///	VRB	XO
PO	BF //	VRA	VRB	XO
PO	RT	EO	VRB	XO
PO	RT	EO MP	VRB	XO
PO	VRT	///	///	XO
PO	VRT	///	VRB	XO
PO	VRT	/// UIM	VRB	XO
PO	VRT	// UIM	VRB	XO
PO	VRT	/ UIM	VRB	XO
PO	VRT	EO	VRB 1 /	XO
PO	VRT	EO	VRB 1 PS	XO
PO	VRT	EO	VRB	XO
PO	VRT	RA	VRB	XO
PO	VRT	SIM	///	XO
PO	VRT	UIM	VRB	XO
PO	VRT	VRA	///	XO
PO	VRT	VRA	VRB 1 /	XO
PO	VRT	VRA	VRB 1 PS	XO
PO	VRT	VRA	VRB	XO

Figure 1.16: VX instruction format

1.6.1.15 X-FORM

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	///	///	///	XO	/
PO	///	///	///	XO	1
PO	///	///	RB	XO	/
PO	///	RA	///	XO	/
PO	///	RA	///	XO	1
PO	///	RA	RB	XO	/
PO	/// L	///	///	XO	/
PO	/// L	///	RB	XO	/
PO	/// 1	RA	RB	XO	/
PO	/// L	RA	RB	XO	Rc
PO	// L	/// SC	///	XO	/
PO	// L	RA	RB	XO	/
PO	/// WC	/// PL	///	XO	/
PO	// IH	///	///	XO	/
PO	/ CT	RA	RB	XO	/
PO	AS //	EO	///	XO	/
PO	AT //	EO	///	XO	/
PO	BF //	///	///	XO	/
PO	BF //	///	FRB	XO	/
PO	BF //	/// W	U /	XO	Rc
PO	BF //	BFA //	///	XO	/
PO	BF //	FRA	FRB	XO	/
PO	BF //	FRA	FRBp	XO	/
PO	BF //	FRAp	FRBp	XO	/
PO	BF //	RA	RB	XO	/
PO	BF //	UIM	FRB	XO	/
PO	BF //	UIM	FRBp	XO	/
PO	BF //	VRA	VRB	XO	/
PO	BF / 1	RA	RB	XO	/
PO	BF / L	RA	RB	XO	/
PO	BF	DCMX	VRB	XO	/
PO	BT	///	///	XO	Rc
PO	D	RA	RB	XO	DX
PO	FRS	RA	RB	XO	/
PO	FRSp	RA	RB	XO	/
PO	FRT	///	FRB	XO	Rc
PO	FRT	///	FRBp	XO	Rc
PO	FRT	EO	///	XO	Rc
PO	FRT	EO	///	XO	/
PO	FRT	EO	/// RM	XO	/
PO	FRT	EO	// DRM	XO	/
PO	FRT	EO	FRB	XO	/
PO	FRT	FRA	FRB	XO	/
PO	FRT	FRA	FRB	XO	Rc
PO	FRT	RA	RB	XO	/
PO	FRT	S ///	FRB	XO	Rc
PO	FRT	SP ///	FRB	XO	Rc
PO	FRTp	///	FRB	XO	Rc
PO	FRTp	///	FRBp	XO	Rc
PO	FRTp	FRA	FRBp	XO	Rc
PO	FRTp	FRAp	FRBp	XO	Rc
PO	FRTp	RA	RB	XO	/
PO	FRTp	S ///	FRBp	XO	Rc
PO	FRTp	SP ///	FRBp	XO	Rc

Figure 1.17: 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				///					RB					XO											/
PO	RS				///					L	///				XO											/
PO	RS				/	RIC				PR	R	RB			XO											/
PO	RS				BFA					//		///			XO											/
PO	RS				RA							///			XO											/
PO	RS				RA							///			XO											1
PO	RS				RA							///			XO											Rc
PO	RS				RA							FC			XO											/
PO	RS				RA							NB			XO											/
PO	RS				RA							SH			XO											Rc
PO	RS				RA							RB			XO											/
PO	RS				RA							RB			XO											1
PO	RS				RA							RB			XO											Rc
PO	RSp				RA							RB			XO											1
PO	RT				///							///			XO											/
PO	RT				///							RB			XO											/
PO	RT				///							RB			XO											1
PO	RT				///					L		///			XO											/
PO	RT				RA							FC			XO											/
PO	RT				RA							NB			XO											/
PO	RT				RA							RB			XO											/
PO	RT				RA							RB			XO											EH
PO	RTp				RA							RB			XO											EH
PO	S				RA							///			XO											SX
PO	S				RA							RB			XO											SX
PO	T				EO							IMM8			XO											TX
PO	T				RA							///			XO											TX
PO	T				RA							RB			XO											TX
PO	TH				RA							RB			XO											/
PO	TO				RA							RB			XO											/
PO	TO				RA							RB			XO											1
PO	VRS				RA							RB			XO											/
PO	VRT				EO							VRB			XO											/
PO	VRT				EO							VRB			XO											RO
PO	VRT				RA							RB			XO											/
PO	VRT				VRA							VRB			XO											/
PO	VRT				VRA							VRB			XO											RO

Figure 1.17: X instruction format

1.6.1.16 XFL-FORM

0	6	7	15	16	21	31
PO	L	FLM	W	FRB	XO	Rc

Figure 1.18: XFL instruction format

1.6.1.17 XFX-FORM

0	6	11	12	15	16	20	21	31
PO	///	///		1	///		XO	/
PO	RS	0	FXM			/	XO	/
PO	RS	1	FXM			/	XO	/
PO	RS	spr					XO	/
PO	RT	0	///			/	XO	/
PO	RT	1	FXM			/	XO	/
PO	RT	BHRBE					XO	/
PO	RT	spr					XO	/
PO	RT	tbr					XO	/

Figure 1.19: XFX instruction format

1.6.1.18 XL-FORM

0	6	9	11	14	16	19	20	21	31
PO	///		///		///			XO	/
PO	///		///		///		S	XO	/
PO	BF	//	BFA	//	///			XO	/
PO	BO		BI		///	BH		XO	LK
PO	BT		BA		BB			XO	/

Figure 1.20: XL instruction format

1.6.1.19 XO-FORM

0	6	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
PO	RT		RA						///					OE	XO									Rc
PO	RT		RA						RB					/	XO									/
PO	RT		RA						RB					/	XO									Rc
PO	RT		RA						RB					OE	XO									Rc

Figure 1.21: XO instruction format

1.6.1.20 XS-FORM

0	6	11	16	21	30	31
PO	RS	RA	sh	XO	sh	Rc

Figure 1.22: XS instruction format

1.6.1.21 XX2-FORM

0	6	9	10	11	12	13	14	15	16	21	25	26	29	30	31
PO	BF	//		///					B	XO				BX	/
PO	BF	DCMX							B	XO				BX	/
PO	RT			EO					B	XO				BX	/
PO	T			///					B	XO				BX	TX
PO	T			///			UIM		B	XO				BX	TX
PO	T	/	UIM						B	XO				BX	TX
PO	T	dx							B	XO	dc	XO	dm	BX	TX
PO	T	EO							B	XO				BX	TX

Figure 1.23: XX2 instruction format

1.6.1.22 XX3-FORM

0	6	9	11	16	21	22	24	29	30	31
PO	AT	//	A	B	XO			AX	BX	/
PO	AT	//	Ap	B	XO			AX	BX	/
PO	BF	//	A	B	XO			AX	BX	/
PO	T		A	B	0	DM	XO	AX	BX	TX
PO	T		A	B	0	SHW	XO	AX	BX	TX
PO	T		A	B	Rc		XO	AX	BX	TX
PO	T		A	B	XO			AX	BX	TX

Figure 1.24: XX3 instruction format

1.6.1.23 XX4-FORM

0	6	11	16	21	26	27	28	29	30	31
PO	T	A	B	C	XO		CX	AX	BX	TX

Figure 1.25: XX4 instruction format

1.6.1.24 Z22-FORM

0	6	9	11	15	16	22	31
PO	BF	//	FRA		DCM	XO	/
PO	BF	//	FRA		DGM	XO	/
PO	BF	//	FRAp		DCM	XO	/
PO	BF	//	FRAp		DGM	XO	/
PO	FRT		FRA		SH	XO	Rc
PO	FRTp		FRAp		SH	XO	Rc

Figure 1.26: Z22 instruction format

1.6.1.25 Z23-FORM

0	6	11	15	16	21	23	31
PO	FRT	///	R	FRB	RMC	XO	Rc
PO	FRT	FRA		FRB	RMC	XO	Rc
PO	FRT	TE		FRB	RMC	XO	Rc
PO	FRTp	///	R	FRBp	RMC	XO	Rc
PO	FRTp	FRA		FRBp	RMC	XO	Rc
PO	FRTp	FRAp		FRBp	RMC	XO	Rc
PO	FRTp	TE		FRBp	RMC	XO	Rc
PO	VRT	///	R	VRB	RMC	XO	/
PO	VRT	///	R	VRB	RMC	XO	EX
PO	RT	RA		RB	CY	XO	/

Figure 1.27: Z23 instruction format

1.6.2 Instruction Prefix Formats

Prefixed instructions consist of a 4-byte prefix followed by a 4-byte suffix. The prefix formats are specified below. The suffix formats share the same formats as word instructions, as specified in Section 1.6.1 on page 11.

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

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

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

0	1	6	8	9	11	12	14	31
1	0	ST	//	R	//	Sub-Type Specific		

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

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

0	1	6	8	9	11	12	14	31
1	0	0	//	R	//	d0		

ST=1: Reserved

1.6.2.2 Type 01 Prefix – Eight-Byte Register-to-Register Instructions

0	1	6	8	12	31
1	1	ST	Sub-type Specific		

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

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

0	1	6	8	12	14	29	31
1	1	0	//	///			
1	1	0	//	///		UIM	
1	1	0	//	///		IMM	
1	1	0	//	//	imm0		

ST=0b0001-0b1111: Reserved

1.6.2.3 Type 10 - Modified Load/Store Instructions

0	1	2	6	8	ST	9	//	11	R	12	//	14	Sub-Type Specific	31

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

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

ST=1: Reserved

1.6.2.4 Type 11 - Modified Register-to-Register Instructions

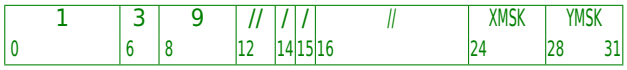
The Type 11 prefix format provides a four-bit subtype (ST) field to specify the subformat employed by the prefix. The subformats are defined as follows.

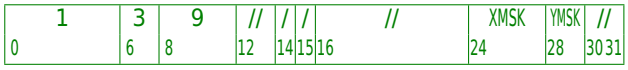
ST=0b0000: Modified Register to Register Form (MRR)

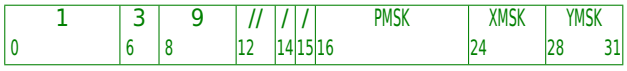
- ***pnop*** (See Section 3.3.20, “Prefixed No-Operation Instruction” on page 132).

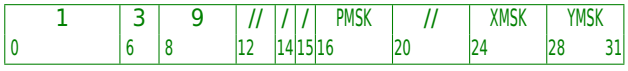
ST=0b0001-0b1000: Reserved

ST=0b1001: Modified Masked Immediate Register to Register Form (MMIRR)









ST=0b1010-0b1111: Reserved

1.6.3 Instruction Fields

This section defines all instruction fields. To distinguish fields in instruction prefixes from fields in word instructions and/or in instruction suffixes, bit numbers for fields in instruction prefixes are preceded by the letter "p". The *Operand* entry indicates what instruction operand is specified by the described field(s) and, for non-identity mappings, how.

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

Operand: This field is not associated with an instruction operand. Different values of this field come from different instruction mnemonics.

AS (6:8)

Field used to specify an Accumulator to be used as a source.

Formats: X

Operand: AS ← AS

AT (6:8)

Field used to specify an Accumulator to be used as a target.

Formats: X, XX3

Operand: AT ← AT

AX,A (29,11:15)

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

Formats: XX3, XX4

Operand: XA ← AX||A

AX,Ap (29,11:15)

Fields that are concatenated to specify an even/odd pair of VSRs to be concatenated and used as a source.

Formats: XX3

Operand: XAp ← AX||Ap

b0,b1,b2 (16:25,11:15,31)

Immediate fields that are concatenated to specify a 16-bit mask.

Formats: DX

Operand: bm ← b0||b1||b2

BA (11:15)

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

Formats: XL

Operand: BA ← BA

BB (16:20)

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

Formats: XL

Operand: BB ← BB

BC (21:25)

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

Formats: A

Operand: BC ← BC

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

Operand: target_addr ← EXTS(BD||0b00)

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

Operand: BF ← BF

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

Operand: BFA ← BFA

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

Operand: BH ← BH

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

Operand: BHRBE ← BHRBE

BI (11:15)

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

Formats: B, XL

Operand: BI ← BI

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

Operand: BO ← BO

BT (6:10)
Field used to specify a bit in the CR or in the FPSCR to be used as a target.
Formats: XL
Operand: BT ← BT

BX,B (30,16:20)
Fields that are concatenated to specify a VSR to be used as a source.
Formats: XX2, XX3, XX4
Operand: XB ← BX||B

CT (7:10)
Field used in X-form instructions to specify a cache target (see Section 4.3.2 of Book II).
Formats: X
Operand: CT ← CT

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

CY (22:23)
Field used to specify a carry mode.
Formats: Z23
Operand: CY ← CY

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
Operand: D ← EXTS(D)

d0,d1 (p14:p31,16:31)
Immediate fields that are concatenated to specify a 34-bit signed two's complement displacement which is sign-extended to 64 bits.
Formats: 8LS, MLS
Operand: D ← EXTS(d0||d1)

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
Operand: D ← EXTS(d0||d1||d2)

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

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

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

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

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

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

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
Operand: disp ← EXTS(DQ||0b0000)

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

DX,D (31,6:10)
Immediate fields that are concatenated to specify a 6-bit unsigned integer which is concatenated on the right with 0b000 and concatenated on the left with 0b111_1111 and sign extended to 64 bits. This 64-bit value is used as a byte offset by the *Hash* instructions.
Formats: X
Operand: offset ← EXTS(0b111_1111||DX||D||0b000)

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
Operand: EH ← EH

EO (11:12)
Expanded opcode field
Formats: X
Operand: This field is not associated with an instruction operand. Different values of this field come from different instruction mnemonics.

EO (11:15)
Expanded opcode field
Formats: VX, X, XX2
Operand: This field is not associated with an instruction operand. Different values of this field come from different instruction mnemonics.

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

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

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

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

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
Operand: FRAp ← FRAp

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

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

FRC (21:25)
Field used to specify an FPR to be used as a source.
Formats: A
Operand: FRC ← FRC

FRS (6:10)
Field used to specify an FPR to be used as a source.
Formats: D, X
Operand: FRS ← FRS

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

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

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
Operand: FRTp ← FRTp

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

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

IMM (11:15)
Immediate field used to select a permute control vector value.
Formats: X
Operand: IMM ← IMM

IMM (p24:p31)
8-bit immediate field used as control operand.
Formats: 8RR
Operand: IMM ← IMM

imm0,imm1 (p16:p31,48:63)
16-bit immediate fields that are concatenated to create a 32-bit value.
Formats: 8RR
Operand: IMM32 ← imm0||imm1

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

IX (14)
Field used to select a doubleword element of a target VSR.
Formats: 8RR:D
Operand: IX ← IX

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

L (8:10)
Field used by the *Data Cache Block Flush* instruction (see Section 4.3.2 of Book II) and also by the *Synchronize* instruction (see Section 4.6.3 of Book II).
Formats: X
Operand: L ← L

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

Field used by the *Paste* instruction to indicate whether to zero the metadata.

Formats: D, X

Operand: L ← L

L (15)

Field used by the *Move To Machine State Register* instruction (see Book III).

Field used by the SLB Invalidate All Global instruction to specify whether the invalidation is for a process or for a partition (see Section 6.9.3.2 of Book III).

Field used by the *SLB Move From Entry VSID* and *SLB Move From Entry ESID* instructions for implementation-specific purposes.

Formats: X

Operand: L ← L

L (14:15)

Field used by the *Deliver A Random Number* instruction (see Section 3.3.9, "Fixed-Point Arithmetic Instructions") to choose the random number format.

Formats: X

Operand: L ← L

LEV (20:26)

Field used by the *System Call* instructions.

Formats: SC

Operand: LEV ← LEV

LI (6:29)

Immediate field used to specify a 24-bit signed two's complement integer which is concatenated on the right with `0b00` and sign-extended to 64 bits.

Formats: I

Operand: target_addr ← EXTS(LI||0b00)

LK (31)

LINK bit.

0 Do not set the Link Register.

1 Set the Link Register. The address of the instruction following the *Branch* instruction is placed into the Link Register.

Formats: B, I, XL

Operand: This field is not associated with an instruction operand. Different values of this field come from different instruction mnemonics.

mb (21:26)

Field used in MD-form and MDS-form instructions to specify the first 1-bit of a 64-bit mask, as described in Section 3.3.14, "Fixed-Point Rotate and Shift Instructions" on page 107.

Formats: MD, MDS

Operand: MB ← $mb_5||mb_{0:4}$

MB (21:25)

Field used in M-form instructions to specify the first 1-bit of a 64-bit mask, as described in Section 3.3.14, "Fixed-Point Rotate and Shift Instructions" on page 107.

Formats: M

Operand: MB ← MB

me (21:26)

Field used in MD-form and MDS-form instructions to specify the last 1-bit of a 64-bit mask, as described in Section 3.3.14, "Fixed-Point Rotate and Shift Instructions" on page 107.

Formats: MD, MDS

Operand: ME ← $me_5||me_{0:4}$

ME (26:30)

Field used in M-form instructions to specify the last 1-bit of a 64-bit mask, as described in Section 3.3.14, "Fixed-Point Rotate and Shift Instructions" on page 107.

Formats: M

Operand: ME ← ME

MP (15)

Field used to specify the bit value to be tested in Vector Count Mask Bits instructions.

Formats: VX

Operand: MP ← MP

N (13:15)

Field used to specify the bit stride in Vector Gather every Nth Bit.

Formats: VX

Operand: N ← N

NB (16:20)

Field used to specify the number of bytes to move in an immediate *Move Assist* instruction.

Formats: X

Operand: NB ← NB

OE (21)

Field used by XO-form instructions to enable setting OV and SO in the XER.

Formats: XO

Operand: This field is not associated with an instruction operand. Different values of this field come from different instruction mnemonics.

PL (14:15)

Field used by the ***wait*** instruction to specify pause length.

Formats: X

Operand: PL ← PL

PMSK (p16:p23)

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

Formats: MMIRR

Operand: PMSK ← PMSK

PMSK (p16:p19)

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

Formats: MMIRR

Operand: PMSK ← PMSK

PMSK (p16:p17)

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

Formats: MMIRR

Operand: PMSK ← PMSK

PO (0:5)

Primary opcode.

Formats: all

Operand: This field is not associated with an instruction operand. Different values of this field come from different instruction mnemonics.

PRS (14)

Field used to specify whether to invalidate process- or partition-scoped entries for ***tlbie***[***l***].

Formats: X

Operand: PRS ← PRS

PS (22)

Field used to specify preferred sign for BCD operations.

Formats: VX

Operand: PS ← PS

R (p11)

Field used to specify whether the effective address of the storage operand is computed relative to the address of the instruction (CIA).

0 Effective address is not computed relative to CIA

1 Effective address is computed relative to CIA

Formats: 8LS, MLS

Operand: R ← R

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***[***l***].

Formats: X, Z23

Operand: R ← R

RA (11:15)

Field used to specify a GPR to be used as a source or as a target.

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

Operand: RA ← RA

RB (16:20)

Field used to specify a GPR to be used as a source.

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

Operand: RB ← RB

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 34.

Formats: VC, XX3

Operand: This field is not associated with an instruction operand. Different values of this field come from different instruction mnemonics.

Rc (31)

RECORD bit.

0 Do not alter the Condition Register.

1 Set Condition Register Field 0 or Field 1 as described in Section 2.3.1, “Condition Register” on page 34.

Formats: A, M, MD, MDS, X, XFL, XO, XS, Z22, Z23

Operand: This field is not associated with an instruction operand. Different values of this field come from different instruction mnemonics.

RC (21:25)

Field used to specify a GPR to be used as a source.

Formats: VA

Operand: RC ← RC

RIC (12:13)

Field used to specify what types of entries to invalidate for ***tlbie***[***l***].

Formats: X

Operand: RIC ← RIC

RM (19:20)

Immediate operand field used to specify new binary floating-point rounding mode.

Formats: X

Operand: RM ← RM

RMC (21:22)

Immediate field used for DFP rounding mode control.

Formats: Z23

Operand: RMC ← RMC

RO (31)

Round to Odd override

Formats: X

Operand: This field is not associated with an instruction operand. Different values of this field come from different instruction mnemonics.

RS (6:10)

Field used to specify a GPR to be used as a source.

Formats: D, DS, M, MD, MDS, X, XFX, XS

Operand: RS ← RS

RSp (6:10)

Field used to specify an even/odd pair of GPRs to be concatenated and used as a source.

Formats: DS, X

Operand: RSp ← RSp

RT (6:10)

Field used to specify a GPR to be used as a target.

Formats: A, D, DS, DX, VA, VX, X, XFX, XO, XX2

Operand: RT ← RT

RTp (6:10)

Field used to specify an even/odd pair of GPRs to be concatenated and used as a target.

Formats: DQ, X

Operand: RTp ← RTp

S (11)

Immediate field that specifies signed versus unsigned conversion.

Formats: X

Operand: S ← S

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

SC (14:15)
Field used by the *Synchronize* instruction to specify the kind(s) of stores that are ordered.
Formats: X
Operand: SC ← SC

sh (16:20,30)
Field containing bits that are concatenated to specify a shift amount.
Formats: MD, XS
Operand: SH ← $sh_5||sh_{0:4}$

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

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

SH (23:25)
Field used to specify a shift amount.
Formats: VN
Operand: SH ← SH

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

SHW (22:23)
Field used to specify a shift amount in words.
Formats: XX3
Operand: SHW ← SHW

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

si0,si1 (p14:p31,16:31)
Immediate fields that are concatenated to specify a 34-bit signed two's complement integer which is sign-extended to 64 bits.
Formats: MLS
Operand: SI ← EXTS(si0||si1)

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

SIX (17:20)
Field used to select the SHA-512 function.
Formats: VX
Operand: SIX ← SIX

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

spr (11:20)
Field used to specify a Special Purpose Register for the ***mtspr*** and ***mfspr*** instructions.
Formats: X
Operand: SPR ← $spr_{5:9}||spr_{0:4}$

ST (16)
Field used to select the SHA-512 function.
Formats: VX
Operand: ST ← ST

SX,S (5,6:9)
Fields SX and S are concatenated to specify a VSR to be used as a source.
Formats: 8LS:D
Operand: XS ← SX||S

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

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

SX,Sp (10,6:9)
Fields that are concatenated to specify an even/odd pair of VRs to be concatenated and used as a source.
Formats: DQ, 8LS:D, X
Operand: XS ← SX||Sp||0b0

tbr (11:20)
Field used by the *Move From Time Base* instruction (see Section 5.1 of Book II).
Formats: X
Operand: TBR ← $tbr_{5:9}||tbr_{0:4}$

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

TH (6:10)
Field used by the data stream variant of the ***dcbt*** and ***dcbtst*** instructions (see Section 4.3.2 of Book II).
Formats: X
Operand: TH ← TH

TO (6:10)
Field used to specify the conditions on which to trap. The encoding is described in Section 3.3.11, “Fixed-Point Trap Instructions” on page 94.
Formats: TX, X
Operand: TO ← TO

TX,T (5,6:10)
Fields that are concatenated to specify a VSR to be used as a target.
Formats: 8LS:D
Operand: XT ← TX||T

TX,T (15,6:10)
Fields that are concatenated to specify a VSR to be used as a target.
Formats: 8RR:D
Operand: XT ← TX||T

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

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
Operand: XT ← TX||T

TX,Tp (10,6:9)
Fields that are concatenated to specify an even/odd pair of VRs to be concatenated and used as a target.
Formats: DQ, 8LS:D, X
Operand: XT ← TX||Tp||0b0

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

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

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

UIM (p29:p31)
3-bit immediate field used as control operand.
Formats: 8RR
Operand: UIM ← UIM

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

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

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

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

UIM (16:20)
Field used to select the special value quadword.
Formats: X
Operand: UIM ← UIM

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

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

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

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

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

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

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
Operand: WC ← WC

XMSK (p24:p27)

Field used to specify ACC row mask for *VSX Vector GER* instructions.

Formats: MMIRR

Operand: XMSK ← XMSK

XO (21,23:31)

Extended opcode field.

Formats: VX

Operand: This field is not associated with an instruction operand. Different values of this field come from different instruction mnemonics.

XO (21:24,26:28)

Extended opcode field.

Formats: XX2

Operand: This field is not associated with an instruction operand. Different values of this field come from different instruction mnemonics.

XO (21:28)

Extended opcode field.

Formats: XX3

Operand: This field is not associated with an instruction operand. Different values of this field come from different instruction mnemonics.

XO (21:29)

Extended opcode field.

Formats: XS, XX2

Operand: This field is not associated with an instruction operand. Different values of this field come from different instruction mnemonics.

XO (21:30)

Extended opcode field.

Formats: X, XFL, XFX, XL

Operand: This field is not associated with an instruction operand. Different values of this field come from different instruction mnemonics.

XO (21:31)

Extended opcode field.

Formats: VX

Operand: This field is not associated with an instruction operand. Different values of this field come from different instruction mnemonics.

XO (22:28)

Extended opcode field.

Formats: XX3

Operand: This field is not associated with an instruction operand. Different values of this field come from different instruction mnemonics.

XO (22:30)

Extended opcode field.

Formats: XO, Z22

Operand: This field is not associated with an instruction operand. Different values of this field come from different instruction mnemonics.

XO (22:31)

Extended opcode field.

Formats: VC

Operand: This field is not associated with an instruction operand. Different values of this field come from different instruction mnemonics.

XO (23:30)

Extended opcode field.

Formats: X, Z23

Operand: This field is not associated with an instruction operand. Different values of this field come from different instruction mnemonics.

XO (24:28)

Extended opcode field.

Formats: XX3

Operand: This field is not associated with an instruction operand. Different values of this field come from different instruction mnemonics.

XO (25:30)

Extended opcode field.

Formats: TX

Operand: This field is not associated with an instruction operand. Different values of this field come from different instruction mnemonics.

XO (26:27)

Extended opcode field.

Formats: XX4

Operand: This field is not associated with an instruction operand. Different values of this field come from different instruction mnemonics.

XO (26:30)

Extended opcode field.

Formats: A, DX

Operand: This field is not associated with an instruction operand. Different values of this field come from different instruction mnemonics.

XO (26:31)

Extended opcode field.

Formats: VA

Operand: This field is not associated with an instruction operand. Different values of this field come from different instruction mnemonics.

XO (27:29)

Extended opcode field.

Formats: MD

Operand: This field is not associated with an instruction operand. Different values of this field come from different instruction mnemonics.

XO (27:30)

Extended opcode field.

Formats: MDS

Operand: This field is not associated with an instruction operand. Different values of this field come from different instruction mnemonics.

XO (29:31)
Extended opcode field.
Formats: DQ
Operand: This field is not associated with an instruction operand. Different values of this field come from different instruction mnemonics.

XO (30)
Extended opcode field.
Formats: SC
Operand: This field is not associated with an instruction operand. Different values of this field come from different instruction mnemonics.

XO (30:31)
Extended opcode field.
Formats: DS, SC
Operand: This field is not associated with an instruction operand. Different values of this field come from different instruction mnemonics.

YMSK (p28:p31)
Field used to specify ACC column mask for *VSX Vector GER* instructions.
Formats: MMIRR
Operand: YMSK ← YMSK

YMSK (p28:p29)
Field used to specify ACC column mask for *VSX Vector GER* instructions.
Formats: MMIRR
Operand: YMSK ← YMSK

Architecture Note

Bits 9:10 for MLS, MMLS, 8LS, and 8MLS prefix formats are reserved for future addition of a cacheability attribute specification for the storage access.

Bits 12:13 for all prefix formats are generally reserved for future addition of a vector length specification for vector instructions. Prefix bits 12:13 would be available for other purposes for non-vector instructions.

1.7 Classes of Instructions

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

- Defined
- Illegal
- Reserved

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

1.7.1 Defined Instruction Class

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

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

1.7.2 Illegal Instruction Class

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

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

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

Architecture Note

Instructions in this class were formerly called “invalid instructions”. The term was changed to “illegal instructions” to reduce confusion between these instructions and invalid *forms* of *defined* instructions.

1.7.3 Reserved Instruction Class

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

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

Any attempt to execute a reserved instruction will:

- perform the actions described by the implementation if the instruction is implemented; or
- cause the system illegal instruction error handler to be invoked if the instruction is not implemented.

1.8 Forms of Defined Instructions

1.8.1 Preferred Instruction Forms

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

Instructions having preferred forms are:

- the *Condition Register Logical* instructions
- the *Load Quadword* instruction
- the *Move Assist* instructions
- the *Or Immediate* instruction (preferred form of no-op)
- the *Move To Condition Register Fields* instruction
- the *Load Quadword and Reserve instruction* (see Section 4.6.2.2 of Book II)

1.8.2 Invalid Instruction Forms

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

In general, any attempt to execute an invalid form of an instruction will either cause the system illegal instruction error handler to be invoked or yield boundedly undefined results. Exceptions to this rule are stated in the instruction descriptions.

Some instruction forms are invalid because the instruction contains a reserved value in a defined field (see Section 1.3.3 on page 4). Instruction forms that are invalid for any other reason are identified in the instruction descriptions.

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

Assembler Note

Assemblers should report uses of invalid instruction forms as errors.

Engineering Note

Causing the system illegal instruction error handler to be invoked if an attempt is made to execute an invalid form of an instruction facilitates the debugging of software.

1.8.3 Reserved-no-op Instructions

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

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

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

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

1.9 Exceptions

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

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

- an attempt to execute an illegal instruction, or an attempt by an application program to execute a "privileged" instruction (see Book III) (system illegal instruction error handler or system privileged instruction error handler)
- the execution of a defined instruction using an invalid form (system illegal instruction error handler or system privileged instruction error handler)
- an attempt to execute an instruction that is not provided by the implementation (system illegal instruction error handler)
- an attempt to execute a prefixed instruction that crosses a 64-byte address boundary. (system alignment error handler)
- an attempt to access a storage location that is unavailable (system instruction storage error handler or system data storage error handler)
- an attempt to access storage with an effective address alignment that is invalid for the instruction (system alignment error handler)
- the execution of a *System Call* or *System Call 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 exceptions that can be caused by an asynchronous event are described in Book III.

The invocation of the system error handler is precise, except that if one of the imprecise modes for invoking the system floating-point enabled exception error handler is in effect (see page 143), then the invocation of the system floating-point enabled exception error handler may also be imprecise. When the system error handler is invoked imprecisely, the excepting instruction does not appear to complete before the next instruction starts (because one of the effects of the excepting instruction, namely the invocation of the system error handler, has not yet occurred).

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

1.10 Storage Addressing

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

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

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

1.10.1 Storage Operands

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

The length of the storage operand is the number of bytes (of the storage operand) that the instruction would access in the absence of invocations of the system error handler. The length is generally implied by the name of the instruction (equivalently, by the opcode, and extended opcode if any). For example, the length of the storage operand of a *Load Word and Zero*, *Load Floating-Point Single*, and *Load Vector Element Word* instruction is four bytes (one word), the length of a *Store Quadword*, *Store Floating-Point Double Pair*, and *Store VSX Vector Word*4* instruction is 16 bytes (one quadword), and the length of a *Load VSX Vector Paired* instruction is 32 bytes (one octword). The only exceptions are the *Load/Store Multiple*, *Move Assist,* and *Load/Store VSX Vector with Length [Left-justified]* 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*), or by the contents of a byte in a GPR (*Load/Store VSX Vector with Length [Left-justified]*), 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.

For most *Load* and *Store* instructions, the storage operand 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 only *Load* and *Store* instructions to which this general rule does not apply are the *Load/Store Multiple* and *Move Assist* instructions, and *Load/Store VSX Vector with Length [Left-justified]* instructions for which the specified length is not an integral power of 2. The storage operand of *Load/Store Multiple Word* is said to be aligned if the address of the storage operand is an integral multiple of four; otherwise the storage operand is said to be unaligned. (The storage operand of a *Move Assist* instruction, or of a *Load/Store VSX Vector with Length [Left-justified]* instruction for which the specified length is not an integral power of 2, 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.)

Operand	Length	$Addr_{58:63}$ if aligned
Byte	8 bits	xxxxxx
Halfword	2 bytes	xxxxx0
Word	4 bytes	xxxx00
Doubleword	8 bytes	xxx000
Quadword	16 bytes	xx0000
Octword	32 bytes	x00000

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 datum 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 datum 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 1.28, unless otherwise specified in the instruction description.

Big-Endian Byte Ordering	
Load	Store
do i = 0 to N-1: $RT_{(R-N)+i}$ ← MEM(EA+i,1)	do i = 0 to N-1: MEM(EA+i,1) ← $(RS)_{(R-N)+i}$
Little-Endian Byte Ordering	
Load	Store
do i = 0 to N-1: $RT_{(R-1)-i}$ ← MEM(EA+i,1)	do i = 0 to N-1: MEM(EA+i,1) ← $(RS)_{(R-1)-i}$
Notes: 1. In this table, subscripts refer to bytes in a register. []	

Figure 1.28: Storage data and byte ordering

Figure 1.29 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 1.30 and 1.31 show each scalar as aligned (assuming an ILP32 ABI). 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 1.30. 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 1.29, are shown in hex (as characters for the elements of the string).

The Little-Endian mapping of structure **s** is shown in Figure 1.31. 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.

```
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;
```

Figure 1.29: C structure ‘s’, showing values of elements

00	11	12	13	14				
	00	01	02	03	04	05	06	07
08	21	22	23	24	25	26	27	28
	08	09	0A	0B	0C	0D	0E	0F
10	31	32	33	34	'A'	'B'	'C'	'D'
	10	11	12	13	14	15	16	17
18	'E'	'F'	'G'		51	52		
	18	19	1A	1B	1C	1D	1E	1F
20	61	62	63	64				
	20	21	22	23	24	25	26	27

Figure 1.30: Big-Endian mapping of structure ‘s’

				11	12	13	14	00
07	06	05	04	03	02	01	00	
21	22	23	24	25	26	27	28	08
0F	0E	0D	0C	0B	0A	09	08	
'D'	'C'	'B'	'A'	31	32	33	34	10
17	16	15	14	13	12	11	10	
		51	52		'G'	'F'	'E'	18
1F	1E	1D	1C	1B	1A	19	18	
				61	62	63	64	20
27	26	25	24	23	22	21	20	

Figure 1.31: Little-Endian mapping of structure ‘s’

1.10.2 Instruction Fetches

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

When an instruction starting at effective address EA is fetched from storage, the relative order of the bytes within each word of the instruction image depends on the byte ordering for the storage access as shown in Figure 1.32.

Big-Endian Byte Ordering
for i=0 to 3: $inst_i$ ← MEM(EA+i,1)
Little-Endian Byte Ordering
for i=0 to 3: $inst_{3-i}$ ← MEM(EA+i,1)
Notes 1. In this table, subscripts refer to bytes within the instruction.

Figure 1.32: Instructions and byte ordering

Figure 1.33 shows an example of a small assembly language program **p**. In the program, prefixed instruction 1 is doubleword-aligned and prefixed instruction 2 is word-aligned.

```
loop:   cmplwi  r5,0
        beq     done
        lwzux   r4,r5,r6
        add     r7,r7,r4
        <prefixed instruction 1>
        subi    r5,r5,4
        <prefixed instruction 2>
        b       loop
done:   stw     r7,total
```

Figure 1.33: Assembly language program ‘p’

The Big-Endian mapping of program **p** is shown in Figure 1.34 (assuming the program starts at address 0).

00	loop: cmplwi r5,0				beq done			
	00	01	02	03	04	05	06	07
08	lwzux r4,r5,r6				add r7,r7,r7			
	08	09	0A	0B	0C	0D	0E	0F
10	<inst 1 prefix>				<inst 1 suffix>			
	10	11	12	13	14	15	16	17
18	subi r5,r5,4				<inst 2 prefix>			
	18	19	1A	1B	1C	1D	1E	1F
20	<inst 2 suffix>				b loop			
	20	21	22	23	24	25	26	27
28	done: stw r7, total							
	28	29	2A	2B	2C	2D	2E	2F

Figure 1.34: Big-Endian mapping of program ‘p’

The Little-Endian mapping of program **p** is shown in Figure 1.35.

beq done				loop: cmplwi r5,0				00
00	01	02	03	04	05	06	07	
add r7,r7,r4				lwzux r4,r5,r6				08
08	09	0A	0B	0C	0D	0E	0F	
<inst 1 suffix>				<inst 1 prefix>				10
10	11	12	13	14	15	16	17	
<inst 2 prefix>				subi r5,r5,4				18
18	19	1A	1B	1C	1D	1E	1F	
b loop				<inst 2 suffix>				20
20	21	22	23	24	25	26	27	
				done: stw r7, total				28
28	29	2A	2B	2C	2D	2E	2F	

Figure 1.35: Little-Endian mapping of program ‘p’

Engineering Note

If the Endian mode changes, e.g. because an ***rfid*** instruction is executed or an interrupt occurs (see Book III), subsequent instructions must be fetched as determined by the new Endian mode regardless of the Endian mode that was in effect when copying instructions into the instruction cache. Implementations that conditionally reverse the order of bytes in instructions when the instructions are copied into the instruction cache must ensure that the instructions are fetched for execution using the correct Endian mode.

Programming Note

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

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

1.10.3 Effective Address Calculation

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

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

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

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

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

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

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

  With X-form instructions that are preceded by an MLS-form prefix with the R bit set to 1 (see Section 1.6.3 of Book I), in computing the effective address of the data element, the contents of the GPR designated by RB are added to the CIA and RA is not used in forming the EA.
- With 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 prefixed instructions having a D-form suffix and an MLS-form or 8LS-form prefix, the 16-bit d1 field is concatenated on the left with the 18-bit d0 field in the prefix, and the concatenation is sign-extended to form a 64-bit address component. In computing the effective address of a data element, this address component is added to the contents of the GPR designated by RA or to zero if RA=0 if the R bit in the prefix is set to 0, or is added to the CIA if the R bit in the prefix is set to 1. (Prefixed instructions corresponding to DS-form and DQ-form word instructions have a D-form suffix.)
- 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 B-form Branch instructions, the 14-bit BD field is concatenated on the right with 0b00 and sign-extended to form a 64-bit address component. If AA=0, this address component is added to the address of the Branch instruction to form the effective address of the target instruction. If AA=1, this address component is the effective address of the target instruction.
- With XL-form Branch instructions, bits 0:61 of the Link Register or the Count Register are concatenated on the right with 0b00 to form the effective address of the target instruction.
- With sequential instruction fetching, if the current instruction is a word instruction, the value 4 is added to the address of the current instruction to form the effective address of the next instruction, and if the current instruction is a prefixed instruction, the value 8 is added to the address of the current instruction to form the effective address of the next instruction, except that if the current instruction is at the maximum instruction effective address for the mode (for a word instruction, 2^{64} - 4 in 64-bit mode and 2^{32} - 4 in 32-bit mode; for a prefixed instruction, 2^{64}-8 in 64-bit mode and 2^{32}-8 in 32-bit mode) the effective address of the next sequential instruction is undefined.

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

Chapter 2. Branch Facility

2.1 Branch Facility Overview

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

2.2 Instruction Execution Order

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

- *Branch* instructions for which the branch is taken cause execution to continue at the target address specified by the Branch instruction.
- *Trap* instructions for which the trap conditions are satisfied, and *System Call* and *System Call Vectored* instructions, cause the appropriate system handler to be invoked.
- Event-based exceptions can cause the event-based branch handler to be invoked, as described in Chapter 6 of Book II.
- Exceptions can cause the system error handler to be invoked, as described in Section 1.9, “Exceptions” on page 26.
- Returning from a system service program, system trap handler, or system error handler causes execution to continue at a specified address.

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

- A floating-point exception occurs when the processor is running in one of the Imprecise floating-point exception modes (see Section 4.4). The instruction that causes the exception need not complete before the next instruction begins execution, with respect to setting exception bits and (if the exception is enabled) invoking the system error handler.
- A *Store* instruction modifies one or more bytes in an area of storage that contains instructions that will subsequently be executed. Before an instruction in that area of storage is executed, software synchronization is required to ensure that the instructions executed are consistent with the results produced by the *Store* instruction.

Programming Note

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

2.3 Branch Facility Registers

2.3.1 Condition Register

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

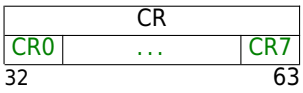


Figure 2.1: Condition Register

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 (***mcrxrx***), or from the FPSCR (***mcrfs***).
- CR Field 0 can be set as the implicit result of a fixed-point instruction.
- CR Field 1 can be set as the implicit result of a floating-point instruction.
- CR Field 1 can be set as the implicit result of a decimal floating-point instruction.
- CR Field 6 can be set as the implicit result of a vector instruction.
- A specified CR field can be set as the result of a *Compare* instruction.

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

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

```
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.

Bit	Definition
0	**Negative** (LT) The result is negative.
1	**Positive** (GT) The result is positive.
2	**Zero** (EQ) The result is zero.
3	**Summary Overflow** (SO) This is a copy of the contents of XER_{SO} at the completion of the instruction.

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

For all floating-point instructions in which Rc=1, CR Field 1 (bits 36:39 of the Condition Register) is set to the Floating-Point exception status, copied from bits 32:35 of the Floating-Point Status and Control Register. This occurs regardless of whether any exceptions are enabled, and regardless of whether the writing of the result is suppressed (see Section 4.4, “Floating-Point Exceptions” on page 142). These bits are interpreted as follows.

Bit	Definition
32	**Floating-Point Exception Summary** (FX) This is a copy of the contents of $FPSCR_{FX}$ at the completion of the instruction.
33	**Floating-Point Enabled Exception Summary** (FEX) This is a copy of the contents of $FPSCR_{FEX}$ at the completion of the instruction.
34	**Floating-Point Invalid Operation Exception Summary** (VX) This is a copy of the contents of $FPSCR_{VX}$ at the completion of the instruction.
35	***Floating-Point Overflow Exception*** (OX) This is a copy of the contents of $FPSCR_{OX}$ at the completion of the instruction.

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 90, and Section 4.6.8, “Floating-Point Compare Instructions” on page 179.

Bit	Definition
0	**Less Than, Floating-Point Less Than** (LT, FL) For fixed-point Compare instructions, (RA) $<$ SI or (RB) (signed comparison) or (RA) $<^{u}$ UI or (RB) (unsigned comparison). For floating-point Compare instructions, (FRA) $<$ (FRB).

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

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

3 **Summary Overflow, Floating-Point Unordered** (SO, FU)
For fixed-point *Compare* instructions, this is a copy of the contents of XER_{SO} 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

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

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.

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

The *Vector Compare Bounds Floating-Point* instruction on page 418 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.

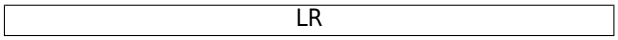


Figure 2.2: Link Register

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 2.3: 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.

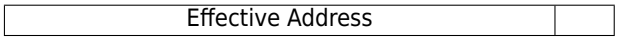


Figure 2.4: Target Address Register

Programming Note

The TAR is reserved for system software.

2.4 Branch Instructions

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

The *Branch* instructions compute the effective address (EA) of the target in one of the following five ways, as described in Section 1.10.3, "Effective Address Calculation" on page 31.

1. Adding a displacement to the address of the *Branch* instruction (*Branch* or *Branch Conditiona*l 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 2.5. In the figure, M=0 in 64-bit mode and M=32 in 32-bit mode.

BO	Description
0000z	Decrement the CTR, then branch if the decremented $CTR_{M:63} \neq 0$ and $CR_{BI}=0$
0001z	Decrement the CTR, then branch if the decremented $CTR_{M:63}=0$ and $CR_{BI}=0$
001at	Branch if $CR_{BI}=0$
0100z	Decrement the CTR, then branch if the decremented $CTR_{M:63} \neq 0$ and $CR_{BI}=1$
0101z	Decrement the CTR, then branch if the decremented $CTR_{M:63}=0$ and $CR_{BI}=1$
011at	Branch if $CR_{BI}=1$
1a00t	Decrement the CTR, then branch if the decremented $CTR_{M:63} \neq 0$
1a01t	Decrement the CTR, then branch if the decremented $CTR_{M:63}=0$
1z1zz	Branch always

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

Figure 2.5: 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 2.6.

at	Hint
00	No hint is given
01	Reserved
10	The branch is very likely not to be taken
11	The branch is very likely to be taken

Figure 2.6: "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 2.7.

BH	Hint
00	***bclr***[***l***]: The instruction is a subroutine return ***bcctr***[***l***] and ***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
01	***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 ***bcctr***[***l***] and ***bctar***[***l***]: Reserved
10	Reserved
11	***bclr***[***l***], ***bcctr***[***l***], and ***bctar***[l]: The target address is not predictable

Figure 2.7: 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).

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.

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

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 ***bcctrl*** 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.

Architecture Note

The bits corresponding to the current “a” and “t” bits, and to the current “z” bits except in the “branch always” BO encoding, had different meanings in versions of the architecture that precede Version 2.00.

- The bit corresponding to the “t” bit was called the “y” bit. The “y” bit indicated whether to use the architected default prediction (y=0) or to use the complement of the default prediction (y=1). The default prediction was defined as follows.
- If the instruction is ***bc***[***l***][***a***] with a negative value in the displacement field, the branch is taken. (This is the only case in which the prediction corresponding to the “y” bit differs from the prediction corresponding to the “t” bit.)
- In all other cases (***bc***[***l***][***a***] with a nonnegative value in the displacement field, ***bclr***[***l***], or ***bcctr***[***l***]), the branch is not taken.
- The BO encodings that test both the Count Register and the Condition Register had a “y” bit in place of the current “z” bit. The meaning of the “y” bit was as described in the preceding item.
- The “a” bit was a “z” bit.

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

Architecture Note

In some future version of the architecture, the value at=0b01 may be used to indicate that the branch path (taken or not taken) is unpredictable (i.e., that neither static nor dynamic prediction is likely to predict the path accurately). It is expected that any new meaning will be such that future *Branch Conditional* instructions that use at=0b01 would use at=0b00 in the current architecture.

Decisions regarding assignment of a meaning for at=0b01 must include consideration of the extent to which software still uses the earlier meaning (see the preceding Architecture Note), and of the effect that the new meaning would have on the performance of such software.

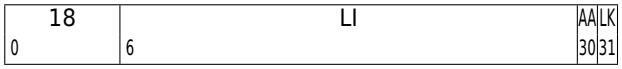
Decisions regarding assignment of a meaning for bit 16 of ***bclr***[***l***] and ***bcctr***[***l***] instructions in some future version of the architecture (e.g., to extend the BH field) must include consideration of the fact that processors that comply with versions of the architecture that precede Version 2.00 may use the bit in computing the prediction associated with the “y” bit. Specifically, for all three *Branch Conditional* instructions, such processors may predict that the branch will be taken if the value of the following expression is 1, and will not be taken if the value is 0. “s” represents bit 16 of the instruction.

$$(BO_0 \ \& \ BO_2) \mid (s \oplus BO_4)$$

The expression assumes that instruction bit 16, which is the sign bit of the displacement field for ***bc***[***l***][***a***], contains 0 for ***bclr***[***l***] and ***bcctr***[***l***].

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)




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

target_addr specifies the branch target address.

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

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

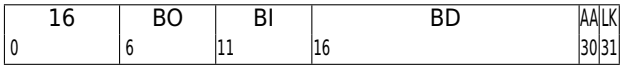
If LK=1 then the effective address of the instruction following the Branch instruction is placed into the Link Register.

Special Registers Altered:

Register	***Field(s)***	
LR		(if 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)




```
if (64-bit mode)
   then M ← 0
   else M ← 32
if ¬BO2 then CTR ← CTR - 1
ctr_ok ← BO2 | ((CTRM:63 ≠ 0) ⊕ BO3)
cond_ok ← BO0 | (CRBI+32 ≡ BO1)
if ctr_ok & cond_ok then
   if AA then NIA ←iea EXTS(BD || 0b00)
   else       NIA ←iea CIA + EXTS(BD || 0b00)
if LK then LR ←iea CIA + 4
```

BI+32 specifies the Condition Register bit to be tested. The BO field is used to resolve the branch as described in Figure 2.5. *target_addr* specifies the branch target address.

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

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

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

Special Registers Altered:

Register	***Field(s)***	
CTR		(if BO_2=0)
LR		(if LK=1)

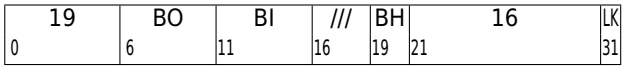
Extended Mnemonics:

Examples of extended mnemonics for *Branch Conditional*:

Extended mnemonic:		**Equivalent to:**	
`blt`	`target`	`bc`	`12,0,target`
`bne`	`cr2,target`	`bc`	`4,10,target`
`bdnz`	`target`	`bc`	`16,0,target`

Branch Conditional to Link Register XL-form

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




```
if (64-bit mode)
   then M ← 0
   else M ← 32
if ¬BO2 then CTR ← CTR - 1
ctr_ok ← BO2 | ((CTRM:63 ≠ 0) ⊕ BO3)
cond_ok ← BO0 | (CRBI+32 ≡ BO1)
if ctr_ok & cond_ok then NIA ←iea LR0:61 || 0b00
if LK then LR ←iea CIA + 4
```

BI+32 specifies the Condition Register bit to be tested. The BO field is used to resolve the branch as described in Figure 2.5. The BH field is used as described in Figure 2.7. 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:

Register	***Field(s)***	
CTR		(if BO_2=0)
LR		(if LK=1)

Extended Mnemonics:

Examples of extended mnemonics for *Branch Conditional to Link Register*:

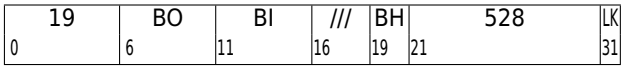
Extended mnemonic:		**Equivalent to:**	
bclr	4,6	bclr	4,6,0
bltlr		bclr	12,0,0
bnelr	cr2	bclr	4,10,0
bdnzlr		bclr	16,0,0

Programming Note

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

Branch Conditional to Count Register XL-form

bcctr	BO,BI,BH	(LK=0)
bcctrl	BO,BI,BH	(LK=1)




```
cond_ok ← BO0 | (CRBI+32 ≡ BO1)
if cond_ok then NIA ←iea CTR0:61 || 0b00
if LK then LR ←iea CIA + 4
```

BI+32 specifies the Condition Register bit to be tested. The BO field is used to resolve the branch as described in Figure 2.5. The BH field is used as described in Figure 2.7. 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 (BO_2=0), the instruction form is invalid.

Special Registers Altered:

Register	***Field(s)***	
LR		(if LK=1)

Extended Mnemonics:

Examples of extended mnemonics for *Branch Conditional to Count Register*:

Extended mnemonic:		**Equivalent to:**	
bcctr	4,6	bcctr	4,6,0
bltctr		bcctr	12,0,0
bnectr	cr2	bcctr	4,10,0

Branch Conditional to Branch Target Address Register XL-form

bctar	BO,BI,BH	(LK=0)
bctarl	BO,BI,BH	(LK=1)

19	BO	BI	///	BH	560	LK
0	6	11	16	19	21	31

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

BI+32 specifies the Condition Register bit to be tested. The BO field is used to resolve the branch as described in Figure 2.5. The BH field is used as described in Figure 2.7. The branch target address is $TAR_{0:61}$ || 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:

Register	*Field(s)*	
CTR		(if BO_2=0)
LR		(if LK=1)

Programming Note

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

2.5 Condition Register Instructions

2.5.1 Condition Register Logical Instructions

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

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

Extended mnemonics for Condition Register logical operations

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

Condition Register AND XL-form

crand BT,BA,BB

19	BT	BA	BB	257	/
0	6	11	16	21	31

$CR_{BT+32} \leftarrow CR_{BA+32}$ & CR_{BB+32}

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

Special Registers Altered:

Register	*Field(s)*
CR	bit BT+32

Condition Register NAND XL-form

crnand BT,BA,BB

19	BT	BA	BB	225	/
0	6	11	16	21	31

$CR_{BT+32} \leftarrow \neg(CR_{BA+32}$ & $CR_{BB+32})$

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

Special Registers Altered:

Register	*Field(s)*
CR	bit BT+32

Condition Register OR XL-form

cror BT,BA,BB

19	BT	BA	BB	449	/
0	6	11	16	21	31

$CR_{BT+32} \leftarrow CR_{BA+32} \mid CR_{BB+32}$

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

Special Registers Altered:

Register	*Field(s)*
CR	bit BT+32

Extended Mnemonics:

Extended Mnemonics for *Condition Register OR*:

Extended mnemonic:	Equivalent to:
`crmove Bx,By`	`cror Bx,By,By`

Condition Register XOR XL-form

crxor BT,BA,BB

19	BT	BA	BB	193	/
0	6	11	16	21	31

$CR_{BT+32} \leftarrow CR_{BA+32} \oplus CR_{BB+32}$

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

Special Registers Altered:

Register	*Field(s)*
CR	bit BT+32

Extended Mnemonics:

Extended Mnemonics for *Condition Register XOR*:

Extended mnemonic:	Equivalent to:
`crclr Bx`	`crxor Bx,Bx,Bx`

Condition Register NOR XL-form

crnor BT,BA,BB

19	BT	BA	BB	33	/
0	6	11	16	21	31

$CR_{BT+32} \leftarrow \neg(CR_{BA+32} \mid CR_{BB+32})$

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

Special Registers Altered:

Register	*Field(s)*
CR	bit BT+32

Extended Mnemonics:

Extended Mnemonics for *Condition Register NOR*:

Extended mnemonic:	Equivalent to:
`crnot Bx,By`	`crnor Bx,By,By`

Condition Register Equivalent XL-form

creqv BT,BA,BB

19	BT	BA	BB	289	/
0	6	11	16	21	31

$CR_{BT+32} \leftarrow CR_{BA+32} \equiv CR_{BB+32}$

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

Special Registers Altered:

Register	*Field(s)*
CR	bit BT+32

Extended Mnemonics:

Extended Mnemonics for *Condition Register Equivalent*:

Extended mnemonic:	Equivalent to:
`crset Bx`	`creqv Bx,Bx,Bx`

Condition Register AND with Complement XL-form

crandc BT,BA,BB

19	BT	BA	BB	129	/
0	6	11	16	21	31

$CR_{BT+32} \leftarrow CR_{BA+32} \ \& \ \neg CR_{BB+32}$

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:

Register	*Field(s)*
CR	bit BT+32

Condition Register OR with Complement XL-form

crorc BT,BA,BB

19	BT	BA	BB	417	/
0	6	11	16	21	31

$CR_{BT+32} \leftarrow CR_{BA+32} \mid \neg CR_{BB+32}$

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.

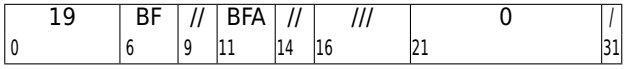
Special Registers Altered:

Register	*Field(s)*
CR	bit BT+32

2.5.2 Condition Register Field Instruction

Move Condition Register Field XL-form

mcrf BF,BFA



$CR_{4\times BF+32:4\times BF+35} \leftarrow CR_{4\times BFA+32:4\times BFA+35}$

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

Special Registers Altered:

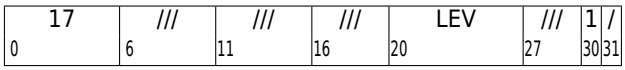
Register	*Field(s)*
CR	field BF

2.6 System Call Instructions

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

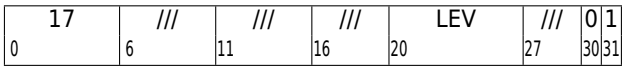
System Call SC-form

sc LEV



System Call Vectored SC-form

scv LEV



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

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

The use of the LEV field is described in Book III. In the first form of the instruction the LEV values greater than 2 are reserved, and bits 0:4 of the LEV field (instruction bits 20:24) are treated as a reserved field.

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

These instructions are context synchronizing (see Book III).

Special Registers Altered:

Dependent on the system service

Programming Note

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

In application programs the value of the LEV operand for ***sc*** should be 0.

Programming Note

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

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 3.1.

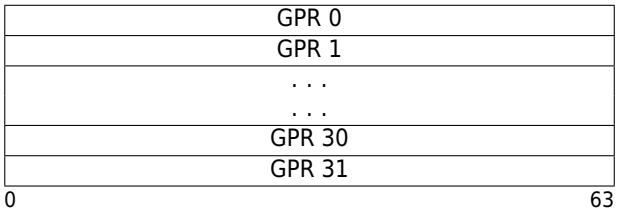


Figure 3.1: General Purpose Registers

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.

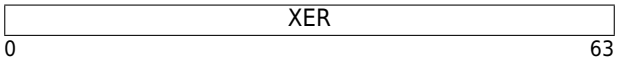


Figure 3.2: Fixed-Point Exception Register

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

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

Bit(s) Definition

0:31 Reserved

32 **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, by ***addex***, or by other instructions (except ***mtspr*** to the XER) that cannot overflow. Executing an ***mtspr*** instruction to the XER, supplying the values 0 for SO and 1 for OV, causes SO to be set to 0 and OV to be set to 1.

33 **Overflow** (OV)

The Overflow bit is set to indicate that an overflow has occurred during execution of an instruction. The Overflow bit can also used as an independent Carry bit by using the ***addex*** with operand 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***, ***divde***, ***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.

34 **Carry** (CA)

The Carry bit is set as follows, during execution of certain instructions. *Add Carrying*, *Subtract From Carrying*, *Add Extended*, and *Subtract From Extended* types of instructions set it to 1 if there is a carry out of bit M, and set it to 0 otherwise. *Shift Right Algebraic* instructions set it to 1 if any 1-bits have been shifted out of a negative operand, and set it to 0 otherwise. The CA bit is not altered by *Compare* instructions, or by other instructions (except *Shift Right Algebraic*, ***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
Bits 48:55 are implemented, and can be read and written by software as if the bits contained a defined field.

57:63 (NBSI)
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. $XER_{48:55}$ 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 1.10.3 on page 31.

Programming Note

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

[]

3.3.1.1 Storage Access Exceptions

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

3.3.2 Fixed-Point Load Instructions

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

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

Programming Note

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

Load Byte and Zero D-form

lbz RT,D(RA)

34	RT	RA	D
0	6	11	16 31

Prefixed Load Byte and Zero MLS:D-form

plbz RT,D(RA),R

Prefix:

1	2	0	//	R	//	d0
0	6	8	9	11	12	14 31

Suffix:

34	RT	RA	d1
0	6	11	16 31

```
if ''lbz'' then
   EA ← (RA|0) + EXTS64(D)
if ''plbz'' & R=0 then
   EA ← (RA|0) + EXTS64(d0||d1)
if ''plbz'' & R=1 then
   EA ← CIA + EXTS64(d0||d1)

RT ← EXTZ(MEM(EA, 1))
```

For ***lbz***, let the effective address (EA) be the sum (RA|0) + EXTS64(D).

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

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

The byte in storage addressed by EA is loaded into $RT_{56:63}$. $RT_{0:55}$ are set to 0.

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

Special Registers Altered:

None

Extended Mnemonics:

Extended Mnemonics for *Prefixed Load Byte and Zero*:

Extended mnemonic:		Equivalent to:	
plbz	RT,D(RA)	plbz	RT,D(RA),0
plbz	RT,D	plbz	RT,D(0),1

Load Byte and Zero Indexed X-form

lbzx RT,RA,RB

31	RT	RA	RB	87	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + (RB)
RT ← 56 0 || MEM(EA, 1)
```

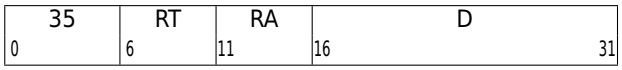
Let the effective address (EA) be the sum (RA|0) + (RB). The byte in storage addressed by EA is loaded into $RT_{56:63}$. $RT_{0:55}$ are set to 0.

Special Registers Altered:

None

Load Byte and Zero with Update D-form

lbzu RT,D(RA)




```
EA ← (RA) + EXTS(D)
RT ← 56 0 || MEM(EA, 1)
RA ← EA
```

Let the effective address (EA) be the sum (RA) + D. The byte in storage addressed by EA is loaded into $RT_{56:63}$. $RT_{0:55}$ are set to 0.

EA is placed into register RA.

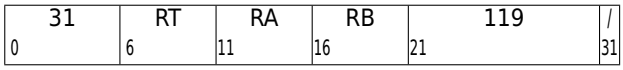
If RA=0 or RA=RT, the instruction form is invalid.

Special Registers Altered:

None

Load Byte and Zero with Update Indexed X-form

lbzux RT,RA,RB




```
EA ← (RA) + (RB)
RT ← 56 0 || MEM(EA, 1)
RA ← EA
```

Let the effective address (EA) be the sum (RA) + (RB). The byte in storage addressed by EA is loaded into $RT_{56:63}$. $RT_{0:55}$ 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 D-form

lhz RT,D(RA)

40	RT	RA	D
0	6	11	16 31

Prefixed Load Halfword and Zero MLS:D-form

plhz RT,D(RA),R

Prefix:

1	2	0	//	R	//	d0
0	6	8	9	11	12	14 31

Suffix:

40	RT	RA	d1
0	6	11	16 31

```
if ''lhz'' then
   EA ← (RA|0) + EXTS64(D)
if ''plhz'' & R=0 then
   EA ← (RA|0) + EXTS64(d0||d1)
if ''plhz'' & R=1 then
   EA ← CIA + EXTS64(d0||d1)

RT ← EXTZ(MEM(EA, 2))
```

For ***lhz***, let the effective address (EA) be the sum (RA|0) + EXTS64(D).

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

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

The halfword in storage addressed by EA is loaded into $RT_{48:63}$. $RT_{0:47}$ are set to 0.

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

Special Registers Altered:

None

Extended Mnemonics:

Extended Mnemonics for *Prefixed Load Halfword and Zero*:

Extended mnemonic:		Equivalent to:	
plhz	RT,D(RA)	plhz	RT,D(RA),0
plhz	RT,D	plhz	RT,D(0),1

Load Halfword and Zero Indexed X-form

lhzx RT,RA,RB

31	RT	RA	RB	279	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + (RB)
RT ← 48^0 || MEM(EA, 2)
```

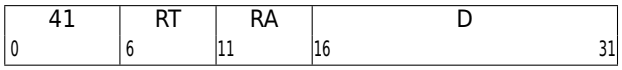
Let the effective address (EA) be the sum (RA|0) + (RB). The halfword in storage addressed by EA is loaded into $RT_{48:63}$. $RT_{0:47}$ are set to 0.

Special Registers Altered:

None

Load Halfword and Zero with Update D-form

lhzu RT,D(RA)




```
EA ← (RA) + EXTS(D)
RT ← 48^0 || 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 set to 0.

EA is placed into register RA.

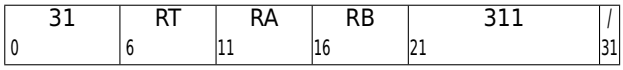
If RA=0 or RA=RT, the instruction form is invalid.

Special Registers Altered:

None

Load Halfword and Zero with Update Indexed X-form

lhzux RT,RA,RB




```
EA ← (RA) + (RB)
RT ← 48^0 || 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 set to 0.

EA is placed into register RA.

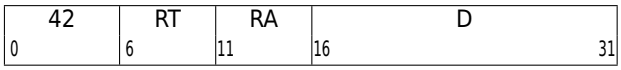
If RA=0 or RA=RT, the instruction form is invalid.

Special Registers Altered:

None

Load Halfword Algebraic D-form

lha RT,D(RA)

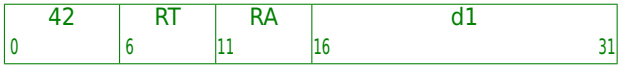


Prefixed Load Halfword Algebraic MLS:D-form

plha RT,D(RA),R

Prefix:

Suffix:




```
if ''lha'' then
   EA ← (RA|0) + EXTS64(D)
if ''plha'' & R=0 then
   EA ← (RA|0) + EXTS64(d0||d1)
if ''plha'' & R=1 then
   EA ← CIA + EXTS64(d0||d1)

RT ← EXTS(MEM(EA, 2))
```

For ***lha***, let the effective address (EA) be the sum (RA|0) + EXTS64(D).

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

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

The 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.

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

Special Registers Altered:

None

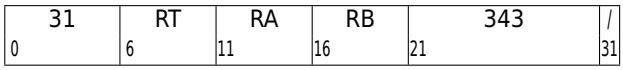
Extended Mnemonics:

Extended Mnemonics for *Prefixed Load Halfword Algebraic*:

Extended mnemonic:		Equivalent to:	
plha	RT,D(RA)	plha	RT,D(RA),0
plha	RT,D	plha	RT,D(0),1

Load Halfword Algebraic Indexed X-form

lhax RT,RA,RB




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

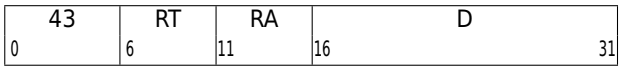
Let the effective address (EA) be the sum (RA|0) + (RB). The halfword in storage addressed by EA is loaded into $RT_{48:63}$. $RT_{0:47}$ are filled with a copy of bit 0 of the loaded halfword.

Special Registers Altered:

None

Load Halfword Algebraic with Update D-form

lhau RT,D(RA)




```
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.

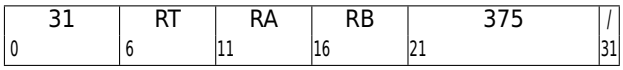
If RA=0 or RA=RT, the instruction form is invalid.

Special Registers Altered:

None

Load Halfword Algebraic with Update Indexed X-form

lhaux RT,RA,RB




```
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.

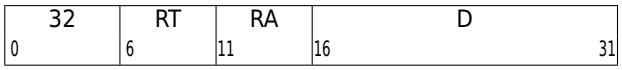
If RA=0 or RA=RT, the instruction form is invalid.

Special Registers Altered:

None

Load Word and Zero D-form

lwz RT,D(RA)

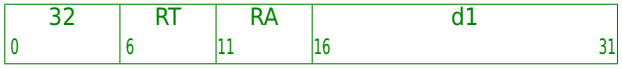


Prefixed Load Word and Zero MLS:D-form

plwz RT,D(RA),R

Prefix:

Suffix:




```
if ''lwz'' then
   EA ← (RA|0) + EXTS64(D)
if ''plwz'' & R=0 then
   EA ← (RA|0) + EXTS64(d0||d1)
if ''plwz'' & R=1 then
   EA ← CIA + EXTS64(d0||d1)

RT ← 32 0 || MEM(EA, 4)
```

For ***lwz***, let the effective address (EA) be the sum (RA|0) + EXTS64(D).

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

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

The word in storage addressed by EA is loaded into $RT_{32:63}$. $RT_{0:31}$ are set to 0.

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

Special Registers Altered:

None

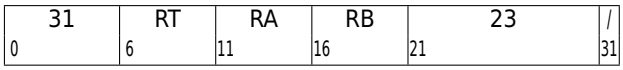
Extended Mnemonics:

Extended Mnemonics for *Prefixed Load Word and Zero*:

Extended mnemonic:		Equivalent to:	
plwz	RT,D(RA)	plwz	RT,D(RA),0
plwz	RT,D	plwz	RT,D(0),1

Load Word and Zero Indexed X-form

lwzx RT,RA,RB




```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + (RB)
RT ← 32 0 || MEM(EA, 4)
```

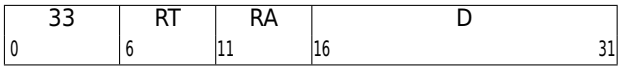
Let the effective address (EA) be the sum (RA|0) + (RB). The word in storage addressed by EA is loaded into $RT_{32:63}$. $RT_{0:31}$ are set to 0.

Special Registers Altered:

None

Load Word and Zero with Update D-form

lwzu RT,D(RA)




```
EA ← (RA) + EXTS(D)
RT ← 32 0 || MEM(EA, 4)
RA ← EA
```

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

EA is placed into register RA.

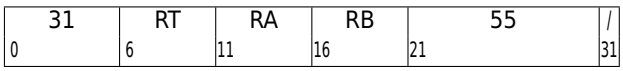
If RA=0 or RA=RT, the instruction form is invalid.

Special Registers Altered:

None

Load Word and Zero with Update Indexed X-form

lwzux RT,RA,RB




```
EA ← (RA) + (RB)
RT ← 32 0 || 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 set to 0.

EA is placed into register RA.

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

Special Registers Altered:

None

3.3.2.1 64-bit Fixed-Point Load Instructions

Load Word Algebraic DS-form

lwa RT,disp(RA)

58	RT	RA	DS	2
0	6	11	16	30 31

Prefixed Load Word Algebraic 8LS:D-form

plwa RT,D(RA),R

Prefix:

1	0	0	//	R	//	d0
0	6	8	9	11	12	14 31

Suffix:

41	RT	RA	d1
0	6	11	16 31

```
if ''lwa'' then
   EA ← (RA|0) + EXTS64(DS||0b00)
if ''plwa'' & R=0 then
   EA ← (RA|0) + EXTS64(d0||d1)
if ''plwa'' & R=1 then
   EA ← CIA + EXTS64(d0||d1)

RT ← EXTS(MEM(EA, 4))
```

For ***lwa***, let the effective address (EA) be the sum (RA|0) + EXTS64(DS||0b00).

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

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

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

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

Special Registers Altered:

None

Extended Mnemonics:

Extended Mnemonics for *Prefixed Load Word Algebraic*:

Extended mnemonic:		Equivalent to:	
plwa	RT,D(RA)	plwa	RT,D(RA),0
plwa	RT,D	plwa	RT,D(0),1

Load Word Algebraic Indexed X-form

lwax RT,RA,RB

31	RT	RA	RB	341	/
0	6	11	16	21	31

```
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 X-form

lwaux RT,RA,RB

31	RT	RA	RB	373	/
0	6	11	16	21	31

```
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

ld RT,disp(RA)

58	RT	RA	DS	0
0	6	11	16	30 31

Prefixed Load Doubleword 8LS:D-form

pld RT,D(RA),R

Prefix:

1	0	0	//	R	//	d0
0	6	8	9	11	12	14 31

Suffix:

57	RT	RA	d1
0	6	11	16 31

```
if ''ld'' then
   EA ← (RA|0) + EXTS64(DS||0b00)
if ''pld'' & R=0 then
   EA ← (RA|0) + EXTS64(d0||d1)
if ''pld'' & R=1 then
   EA ← CIA + EXTS64(d0||d1)

RT ← MEM(EA, 8)
```

For ***ld***, let the effective address (EA) be the sum (RA|0) + EXTS64(DS||0b00).

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

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

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

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

Special Registers Altered:

None

Extended Mnemonics:

Extended Mnemonics for *Prefixed Load Doubleword*:

Extended mnemonic:		Equivalent to:	
pld	RT,D(RA)	pld	RT,D(RA),0
pld	RT,D	pld	RT,D(0),1

Load Doubleword Indexed X-form

ldx RT,RA,RB

31	RT	RA	RB	21	/
0	6	11	16	21	31

```
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

ldu RT,disp(RA)

58	RT	RA	DS	1
0	6	11	16	30 31

```
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

ldux RT,RA,RB

31	RT	RA	RB	53	/
0	6	11	16	21	31

```
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

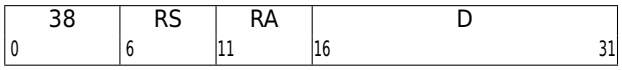
The contents of register RS are stored into the byte, halfword, word, or doubleword in storage addressed by EA.

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.

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

Store Byte D-form

stb RS,D(RA)

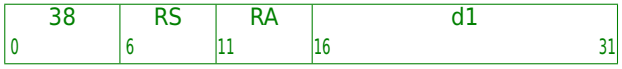


Prefixed Store Byte MLS:D-form

pstb RS,D(RA),R

Prefix:

Suffix:




```
if ''stb'' then
   EA ← (RA|0) + EXTS64(D)
if ''pstb'' & R=0 then
   EA ← (RA|0) + EXTS64(d0||d1)
if ''pstb'' & R=1 then
   EA ← CIA + EXTS64(d0||d1)

MEM(EA, 1) ← (RS)56:63
```

For ***stb***, let the effective address (EA) be the sum (RA|0) + EXTS64(D).

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

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

$(RS)_{56:63}$ are stored into the byte in storage addressed by EA.

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

Special Registers Altered:

None

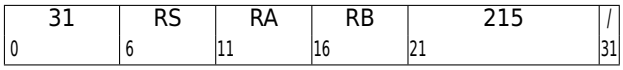
Extended Mnemonics:

Extended Mnemonics for *Prefixed Store Byte*:

Extended mnemonic:		Equivalent to:	
pstb	RS,D(RA)	pstb	RS,D(RA),0
pstb	RS,D	pstb	RS,D(0),1

Store Byte Indexed X-form

stbx RS,RA,RB




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

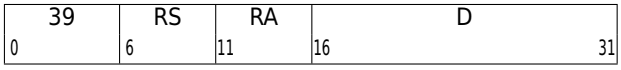
Let the effective address (EA) be the sum (RA|0) + (RB). $(RS)_{56:63}$ are stored into the byte in storage addressed by EA.

Special Registers Altered:

None

Store Byte with Update D-form

stbu RS,D(RA)




```
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.

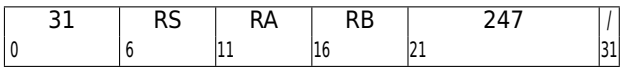
If RA=0, the instruction form is invalid.

Special Registers Altered:

None

Store Byte with Update Indexed X-form

stbux RS,RA,RB




```
EA ← (RA) + (RB)
MEM(EA, 1) ← (RS)56:63
RA ← EA
```

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

EA is placed into register RA.

If RA=0, the instruction form is invalid.

Special Registers Altered:

None

Store Halfword D-form

sth RS,D(RA)

44	RS	RA	D
0	6	11	16 31

Prefixed Store Halfword MLS:D-form

psth RS,D(RA),R

Prefix:

1	2	0	//	R	//	d0
0	6	8	9	11	12	14 31

Suffix:

44	RS	RA	d1
0	6	11	16 31

```
if ''sth'' then
   EA ← (RA|0) + EXTS64(D)
if ''psth'' & R=0 then
   EA ← (RA|0) + EXTS64(d0||d1)
if ''psth'' & R=1 then
   EA ← CIA + EXTS64(d0||d1)

MEM(EA, 2) ← (RS)48:63
```

For ***sth***, let the effective address (EA) be the sum (RA|0) + EXTS64(D).

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

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

$(RS)_{48:63}$ are stored into the halfword in storage addressed by EA.

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

Special Registers Altered:

None

Extended Mnemonics:

Extended Mnemonics for *Prefixed Store Halfword*:

Extended mnemonic:		Equivalent to:	
psth	RS,D(RA)	psth	RS,D(RA),0
psth	RS,D	psth	RS,D(0),1

Store Halfword Indexed X-form

sthx RS,RA,RB

31	RS	RA	RB	407	/
0	6	11	16	21	31

```
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.

Special Registers Altered:

None

Store Halfword with Update D-form

sthu RS,D(RA)

45	RS	RA	D
0	6	11	16 31

```
EA ← (RA) + EXTS(D)
MEM(EA, 2) ← (RS)48:63
RA ← EA
```

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

EA is placed into register RA.

If RA=0, the instruction form is invalid.

Special Registers Altered:

None

Store Halfword with Update Indexed X-form

sthux RS,RA,RB

31	RS	RA	RB	439	/
0	6	11	16	21	31

```
EA ← (RA) + (RB)
MEM(EA, 2) ← (RS)48:63
RA ← EA
```

Let the effective address (EA) be the sum (RA) + (RB). $(RS)_{48:63}$ are stored into the halfword in storage addressed by EA.

EA is placed into register RA.

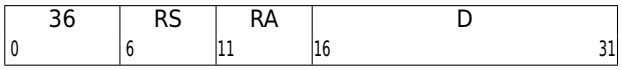
If RA=0, the instruction form is invalid.

Special Registers Altered:

None

Store Word D-form

stw RS,D(RA)

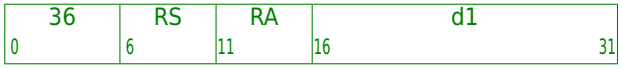


Prefixed Store Word MLS:D-form

pstw RS,D(RA),R

Prefix:

Suffix:




```
if ''stw'' then
   EA ← (RA|0) + EXTS64(D)
if ''pstw'' & R=0 then
   EA ← (RA|0) + EXTS64(d0||d1)
if ''pstw'' & R=1 then
   EA ← CIA + EXTS64(d0||d1)

MEM(EA, 4) ← (RS)32:63
```

For ***stw***, let the effective address (EA) be the sum (RA|0) + EXTS64(D).

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

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

$(RS)_{32:63}$ are stored into the word in storage addressed by EA.

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

Special Registers Altered:

None

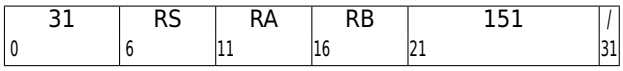
Extended Mnemonics:

Extended Mnemonics for *Prefixed Store Word*:

Extended mnemonic:		Equivalent to:	
pstw	RS,D(RA)	pstw	RS,D(RA),0
pstw	RS,D	pstw	RS,D(0),1

Store Word Indexed X-form

stwx RS,RA,RB




```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + (RB)
MEM(EA, 4) ← (RS)32:63
```

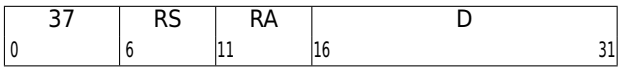
Let the effective address (EA) be the sum (RA|0) + (RB). $(RS)_{32:63}$ are stored into the word in storage addressed by EA.

Special Registers Altered:

None

Store Word with Update D-form

stwu RS,D(RA)




```
EA ← (RA) + EXTS(D)
MEM(EA, 4) ← (RS)32:63
RA ← EA
```

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

EA is placed into register RA.

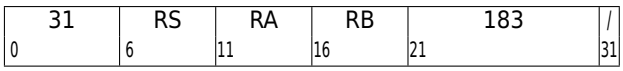
If RA=0, the instruction form is invalid.

Special Registers Altered:

None

Store Word with Update Indexed X-form

stwux RS,RA,RB




```
EA ← (RA) + (RB)
MEM(EA, 4) ← (RS)32:63
RA ← EA
```

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

EA is placed into register RA.

If RA=0, the instruction form is invalid.

Special Registers Altered:

None

3.3.3.1 64-bit Fixed-Point Store Instructions

Store Doubleword DS-form

std RS,disp(RA)

62	RS	RA	DS	0
0	6	11	16	30 31

Prefixed Store Doubleword 8LS:D-form

pstd RS,D(RA),R

Prefix:

1	0	0	//	R	//	d0
0	6	8	9	11	12	14 31

Suffix:

61	RS	RA	d1
0	6	11	16 31

```
if ''std'' then
   EA ← (RA|0) + EXTS64(DS||0b00)
if ''pstd'' & R=0 then
   EA ← (RA|0) + EXTS64(d0||d1)
if ''pstd'' & R=1 then
   EA ← CIA + EXTS64(d0||d1)

MEM(EA, 8) ← (RS)
```

For ***std***, let the effective address (EA) be the sum (RA|0) + EXTS64(DS||0b00).

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

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

(RS) is stored into the doubleword in storage addressed by EA.

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

Special Registers Altered:

None

Extended Mnemonics:

Extended Mnemonics for *Prefixed Store Doubleword*:

Extended mnemonic:		Equivalent to:	
pstd	RS,D(RA)	pstd	RS,D(RA),0
pstd	RS,D	pstd	RS,D(0),1

Store Doubleword Indexed X-form

stdx RS,RA,RB

31	RS	RA	RB	149	/
0	6	11	16	21	31

```
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).

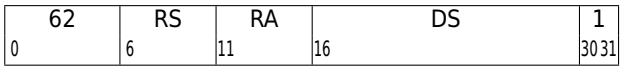
(RS) is stored into the doubleword in storage addressed by EA.

Special Registers Altered:

None

Store Doubleword with Update DS-form

stdu RS,disp(RA)




```
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.

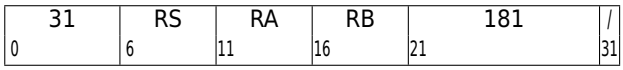
If RA=0, the instruction form is invalid.

Special Registers Altered:

None

Store Doubleword with Update Indexed X-form

stdux RS,RA,RB




```
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.

On the other hand, for ***plq***, the quadword in storage addressed by EA is loaded into an even-odd pair of GPRs as follows. Independent of endian mode, the even-numbered GPR is loaded with the doubleword from storage addressed by EA and the odd-numbered GPR is loaded with the doubleword addressed by EA+8.

In the preferred form of the *Load Quadword* instruction RA $\neq$ RTp+1.

For ***stq***, the contents of an even-odd pair of GPRs is stored into the quadword in storage addressed by EA as follows. In Big-Endian mode, the even-numbered GPR is stored into the doubleword in storage addressed by EA and the odd-numbered GPR is stored into the doubleword addressed by EA+8. In Little-Endian mode, the even-numbered GPR is stored byte-reversed into the doubleword in storage addressed by EA+8 and the odd-numbered GPR is stored byte-reversed into the doubleword addressed by EA.

On the other hand, for ***pstq***, the contents of an even-odd pair of GPRs is stored into the quadword in storage addressed by EA as follows. Independent of endian mode, the even-numbered GPR is stored into the doubleword in storage addressed by EA and the odd-numbered GPR is stored into the doubleword addressed by EA+8.

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 DQ-form

lq RTp,disp(RA)

56	RTp	RA	DQ	///
0	6	11	16	28 31

Prefixed Load Quadword 8LS:D-form

plq RTp,D(RA),R

Prefix:

1	0	0	//	R	//	d0
0	6	8	9	11	12	14 31

Suffix:

56	RTp	RA	d1
0	6	11	16 31

```
if ''lq'' then
   EA ← (RA|0) + EXTS64(DQ||0b0000)
if ''plq'' & R=0 then
   EA ← (RA|0) + EXTS64(d0||d1)
if ''plq'' & R=1 then
   EA ← CIA + EXTS64(d0||d1)
if Big-Endian byte ordering then
   RTp||RTp+1 ← MEM(EA,16)
if ''lq'' and Little-Endian byte ordering then
   RTp||RTp+1 ← MEM(EA,16)
if ''plq'' and Little-Endian byte ordering then
   RTp+1||RTp ← MEM(EA,16)
```

For ***lq***, let the effective address (EA) be the sum of the contents of register RA, or the value 0 if RA=0, and the value DQ||0b0000, sign-extended to 64 bits.

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

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

For Big-Endian byte ordering, the quadword in storage addressed by EA is loaded into RTp||RTp+1.

For ***lq*** and Little-Endian byte ordering, the quadword in storage addressed by EA is byte-reversed and loaded into RTp||RTp+1.

For ***plq*** and Little-Endian byte ordering, the quadword in storage addressed by EA is byte-reversed and loaded into RTp+1||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.)

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

Programming Note

In versions of the architecture prior to v2.07, this instruction was privileged.

Special Registers Altered:

None

Extended Mnemonics:

Extended Mnemonics for *Prefixed Load Quadword*:

Extended mnemonic:		Equivalent to:	
plq	RTp,D(RA)	plq	RTp,D(RA),0
plq	RTp,D	plq	RTp,D(0),1

Store Quadword DS-form

stq RSp,disp(RA)

62	RSp	RA	DS	2
0	6	11	16	30 31

Prefixed Store Quadword 8LS:D-form

pstq RSp,D(RA),R

Prefix:

1	0	0	//	R	//	d0
0	6	8	9	11	12	14 31

Suffix:

60	RSp	RA	d1
0	6	11	16 31

```
if ''stq'' then
   EA ← (RA|0) + EXTS64(DS||0b00)
if ''pstq'' & R=0 then
   EA ← (RA|0) + EXTS64(d0||d1)
if ''pstq'' & R=1 then
   EA ← CIA + EXTS64(d0||d1)
if Big-Endian byte ordering then
   MEM(EA,16) ← (RSp)||(RSp+1)
if ''stq'' and Little-Endian byte ordering then
   MEM(EA,16) ← (RSp)||(RSp+1)
if ''pstq'' and Little-Endian byte ordering then
   MEM(EA,16) ← (RSp+1)||(RSp)
```

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

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

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

For Big-Endian byte ordering, the content of register pair RSp||RSp+1 is stored into the quadword in storage addressed by EA.

For ***stq*** and Little-Endian byte ordering, the content of register pair RSp||RSp+1 is byte-reversed and stored into the quadword in storage addressed by EA.

For ***pstq*** and Little-Endian byte ordering, the content of register pair RSp+1||RSp is byte-reversed and stored into the quadword in storage addressed by EA.

If RSp is odd, the instruction form is invalid.

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

Programming Note

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

Special Registers Altered:

None

Extended Mnemonics:

Extended Mnemonics for *Prefixed Store Quadword*:

Extended mnemonic:		Equivalent to:	
pstq	RSp,D(RA)	pstq	RSp,D(RA),0
pstq	RSp,D	pstq	RSp,D(0),1

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.

Programming Note

In some implementations, the *Load Byte-Reverse* instructions may have greater latency than other *Load* instructions.

Load Halfword Byte-Reverse Indexed X-form

lhbrx RT,RA,RB

31	RT	RA	RB	790	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + (RB)
load_data ← MEM(EA, 2)
RT ← 48^0 || load_data8:15 || load_data0:7
```

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

Bits 0:7 of the halfword in storage addressed by EA are loaded into $RT_{56:63}$.

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

sthbrx RS,RA,RB

31	RS	RA	RB	918	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + (RB)
MEM(EA, 2) ← (RS)56:63 || (RS)48:55
```

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

$(RS)_{56:63}$ are stored into bits 0:7 of the halfword in storage addressed by EA.

$(RS)_{48:55}$ are stored into bits 8:15 of the halfword in storage addressed by EA.

Special Registers Altered:

None

Load Word Byte-Reverse Indexed X-form

lwbrx RT,RA,RB

31	RT	RA	RB	534	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + (RB)
load_data ← MEM(EA, 4)
RT ← 32^0 || load_data24:31 || load_data16:23
        || load_data8:15 || load_data0:7
```

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 $RT_{40:47}$.

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

stwbrx RS,RA,RB

31	RS	RA	RB	662	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← 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.

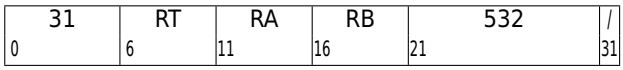
Special Registers Altered:

None

3.3.5.1 64-Bit Load and Store with Byte Reversal Instructions

Load Doubleword Byte-Reverse Indexed X-form

ldbrx RT,RA,RB




```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + (RB)
load_data ← MEM(EA, 8)
RT ←  load_data56:63 || load_data48:55
        || load_data40:47 || load_data32:39
        || load_data24:31 || load_data16:23
        || load_data8:15  || load_data0:7
```

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 $RT_{56:63}$.

Bits 8:15 of the doubleword in storage addressed by EA are loaded into $RT_{48:55}$.

Bits 16:23 of the doubleword in storage addressed by EA are loaded into $RT_{40:47}$.

Bits 24:31 of the doubleword in storage addressed by EA are loaded into $RT_{32:39}$.

Bits 32:39 of the doubleword in storage addressed by EA are loaded into $RT_{24:31}$.

Bits 40:47 of the doubleword in storage addressed by EA are loaded into $RT_{16:23}$.

Bits 48:55 of the doubleword in storage addressed by EA are loaded into $RT_{8:15}$.

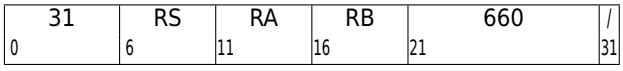
Bits 56:63 of the doubleword in storage addressed by EA are loaded into $RT_{0:7}$.

Special Registers Altered:

None

Store Doubleword Byte-Reverse Indexed X-form

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
               || (RS)40:47 || (RS)32:39
               || (RS)24:31 || (RS)16:23
               || (RS)8:15  || (RS)0:7
```

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

$(RS)_{56:63}$ are stored into bits 0:7 of the doubleword in storage addressed by EA.

$(RS)_{48:55}$ are stored into bits 8:15 of the doubleword in storage addressed by EA.

$(RS)_{40:47}$ are stored into bits 16:23 of the doubleword in storage addressed by EA.

$(RS)_{32:39}$ are stored into bits 23:31 of the doubleword in storage addressed by EA.

$(RS)_{24:31}$ are stored into bits 32:39 of the doubleword in storage addressed by EA.

$(RS)_{16:23}$ are stored into bits 40:47 of the doubleword in storage addressed by EA.

$(RS)_{8:15}$ are stored into bits 48:55 of the doubleword in storage addressed by EA.

$(RS)_{0:7}$ are stored into bits 56:63 of the doubleword in storage addressed by EA.

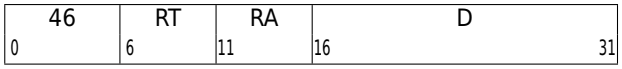
Special Registers Altered:

None

3.3.6 Fixed-Point Load and Store Multiple Instructions

Load Multiple Word D-form

lmw RT,D(RA)




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

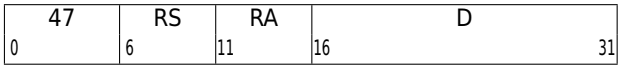
This instruction is not supported in Little-Endian mode. If it is executed in Little-Endian mode, the system alignment error handler is invoked.

Special Registers Altered:

None

Store Multiple Word D-form

stmw RS,D(RA)




```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + EXTS(D)
r ← RS
do while r ≤ 31
   MEM(EA, 4) ← GPR(r)32:63
   r ← r + 1
   EA ← EA + 4
```

Let n = (32−RS). Let the effective address (EA) be the sum (RA|0) + D.

n consecutive words starting at EA are stored from the low-order 32 bits of GPRs RS through 31.

This instruction is not supported in Little-Endian mode. If it is executed in Little-Endian mode, the system alignment error handler is invoked.

Special Registers Altered:

None

3.3.7 Fixed-Point Move Assist Instructions [Phased Out]

The *Move Assist* instructions allow movement of an arbitrary sequence of bytes from storage to registers or from registers to storage without concern for alignment. These instructions can be used for a short move between arbitrary storage locations or to initiate a long move between unaligned storage fields.

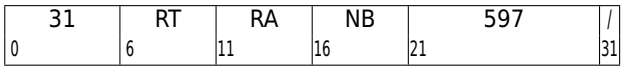
The *Move Assist* instructions have preferred forms; see Section 1.8.1, “Preferred Instruction Forms” on page 25. In the preferred forms, register usage satisfies the following rules.

- RS = 4 or 5
- RT = 4 or 5
- last register loaded/stored $\leq$ 12

[]

Load String Word Immediate X-form

lswi RT,RA,NB




```
if RA = 0 then EA ← 0
else          EA ← (RA)
if NB = 0 then n ← 32
else          n ← NB
r ← RT - 1
i ← 32
do while n > 0
   if i = 32 then
      r ← r + 1 (mod 32)
      GPR(r) ← 0
   GPR(r)i:i+7 ← MEM(EA, 1)
   i ← i + 8
   if i = 64 then i ← 32
   EA ← EA + 1
   n ← n - 1
```

Let the effective address (EA) be (RA|0). Let n = NB if NB≠0, n = 32 if NB=0; n is the number of bytes to load. Let nr=CEIL(n/4); nr is the number of registers to receive data.

n consecutive bytes starting at EA are loaded into GPRs RT through RT+nr-1. Data are loaded into the low-order four bytes of each GPR; the high-order four bytes are set to 0.

Bytes are loaded left to right in each register. The sequence of registers wraps around to GPR 0 if required. If the low-order four bytes of register RT+nr-1 are only partially filled, the unfilled low-order byte(s) of that register are set to 0.

If RA is in the range of registers to be loaded, including the case in which RA=0, the instruction form is invalid.

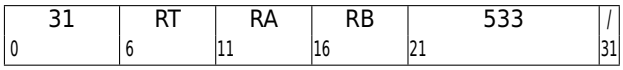
This instruction is not supported in Little-Endian mode. If it is executed in Little-Endian mode, the system alignment error handler is invoked.

Special Registers Altered:

None

Load String Word Indexed X-form

lswx RT,RA,RB




```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + (RB)
n ← XER57:63
r ← RT - 1
i ← 32
RT ← undefined
do while n > 0
   if i = 32 then
      r ← r + 1 (mod 32)
      GPR(r) ← 0
   GPR(r)i:i+7 ← MEM(EA, 1)
   i ← i + 8
   if i = 64 then i ← 32
   EA ← EA + 1
   n ← n - 1
```

Let the effective address (EA) be the sum (RA|0)+ (RB). Let n=XER57:63; n is the number of bytes to load. Let nr=CEIL(n/4); nr is the number of registers to receive data.

If n>0, n consecutive bytes starting at EA are loaded into GPRs RT through RT+nr-1. Data are loaded into the low-order four bytes of each GPR; the high-order four bytes are set to 0.

Bytes are loaded left to right in each register. The sequence of registers wraps around to GPR 0 if required. If the low-order four bytes of register RT+nr-1 are only partially filled, the unfilled low-order byte(s) of that register are set to 0.

If n=0, the contents of register RT are undefined.

If RA or RB is in the range of registers to be loaded, including the case in which RA=0, the instruction is treated as if the instruction form were invalid. If RT=RA or RT=RB, the instruction form is invalid.

This instruction is not supported in Little-Endian mode. If it is executed in Little-Endian mode and n>0, the system alignment error handler is invoked.

Special Registers Altered:

None

Store String Word Immediate X-form

stswi RS,RA,NB

31	RS	RA	NB	725	/
0	6	11	16	21	31

```
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

stswx RS,RA,RB

31	RS	RA	RB	661	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + (RB)
n ← XER57:63
r ← RS - 1
i ← 32
do while n > 0
   if i = 32 then r ← r + 1 (mod 32)
   MEM(EA, 1) ← GPR(r)i:i+7
   i ← i + 8
   if i = 64 then i ← 32
   EA ← EA + 1
   n ← n - 1
```

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***, ***divde***, ***divdu*** and ***divdeu***, and overflow of the low-order 32-bit result for ***mullw***, ***divw***, ***divwe***, ***divwu***, and ***divweu***.

Programming Note

Notice that CR Field 0 may not reflect the “true” (infinitely precise) result if overflow occurs.

Extended mnemonics for addition and subtraction

Several extended mnemonics are provided that use the *Add Immediate* and *Add Immediate Shifted* instructions to load an immediate value or an address into a target register. Some of these are shown as examples with the two instructions.

The Power ISA supplies *Subtract From* instructions, which subtract the second operand from the third. A set of extended mnemonics is provided that use the more “normal” order, in which the third operand is subtracted from the second, with the third operand being either an immediate field or a register. Some of these are shown as examples with the appropriate *Add* and *Subtract From* instructions.

See Appendix C for additional extended mnemonics.

Add Immediate D-form

addi RT,RA,SI

14	RT	RA	SI
0	6	11	16 31

Prefixed Add Immediate MLS:D-form

paddi RT,RA,SI,R

Prefix:

1	2	0	//	R	//	si0
0	6	8	9	11	12	14 31

Suffix:

14	RT	RA	si1
0	6	11	16 31

```
if ''addi'' then
   RT ← (RA|0) + EXTS64(SI)
if ''paddi'' & R=0 then
   RT ← (RA|0) + EXTS64(si0||si1)
if ''paddi'' & R=1 then
   RT ← CIA + EXTS64(si0||si1)
```

For ***addi***, the sum of the contents of register RA, or the value 0 if RA=0, and the value SI, sign-extended to 64 bits, is placed into register RT.

For ***paddi*** with R=0, the sum of the contents of register RA, or the value 0 if RA=0, and the value si0||si1, sign-extended to 64 bits, is placed into register RT.

For ***paddi*** with R=1, the sum of the address of the instruction and the value si0||si1, sign-extended to 64 bits, is placed into register RT.

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

Special Registers Altered:

None

Extended Mnemonics:

Extended Mnemonics for *Add Immediate*:

Extended mnemonic:		Equivalent to:	
la	RT,SI(RA)	addi	RT,RA,SI
li	RT,SI	addi	RT,0,SI
subi	RT,RA,si	addi	RT,RA,-si

Examples of extended mnemonics for *Prefixed Add Immediate*:

Extended mnemonic:		Equivalent to:	
paddi	RT,RA,SI	paddi	RT,RA,SI,0
pla	RT,SI(RA)	paddi	RT,RA,SI,0
pla	RT,SI	paddi	RT,0,SI,1
pli	RT,SI	paddi	RT,0,SI,0
psubi	RT,RA,si	paddi	RT,RA,-si,0

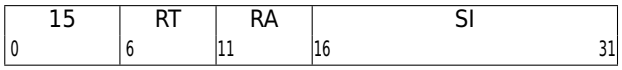
Programming Note

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 D-form

addis RT,RA,SI




```
if RA = 0 then RT ← EXTS(SI || 16 0)
else           RT ← (RA) + EXTS(SI || 16 0)
```

The sum `(RA|0) + (SI || 0x0000)` is placed into register `RT`.

Special Registers Altered:

None

Extended Mnemonics:

Extended Mnemonics for *Add Immediate Shifted*:

Extended mnemonic:		Equivalent to:	
lis	RT,SI	addis	RT,0,SI
subis	RT,RA,si	addis	RT,RA,-si

Add PC Immediate Shifted DX-form

addpcis RT,D


```
D ← d0||d1||d2
RT ← NIA + EXTS(D || 16 0)
```

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

Special Registers Altered:

None

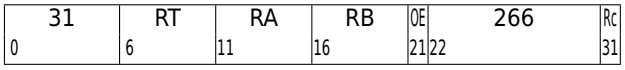
Extended Mnemonics:

Extended Mnemonics for *Add PC Immediate Shifted*:

Extended mnemonic:		Equivalent to:	
lnia	RT	addpcis	RT,0
subpcis	RT,d	addpcis	RT,-d

Add XO-form

add	RT,RA,RB	(OE=0 Rc=0)
add.	RT,RA,RB	(OE=0 Rc=1)
addo	RT,RA,RB	(OE=1 Rc=0)
addo.	RT,RA,RB	(OE=1 Rc=1)




```
RT ← (RA) + (RB)
```

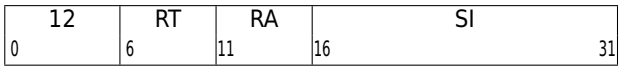
The sum `(RA) + (RB)` is placed into register `RT`.

Special Registers Altered:

Register	*Field(s)*	
CR	CR0	(if Rc=1)
XER	SO OV OV32	(if OE=1)

Add Immediate Carrying D-form

addic RT,RA,SI




```
RT ← (RA) + EXTS(SI)
```

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

Special Registers Altered:

Register	*Field(s)*
XER	CA CA32

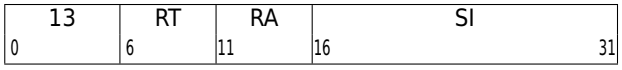
Extended Mnemonics:

Extended Mnemonics for *Add Immediate Carrying*:

Extended mnemonic:		Equivalent to:	
`subic`	`RT,RA,si`	`addic`	`RT,RA,-si`

Add Immediate Carrying and Record D-form

addic. RT,RA,SI




```
RT ← (RA) + EXTS(SI)
```

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

Special Registers Altered:

Register	*Field(s)*
CR	CR0
XER	CA CA32

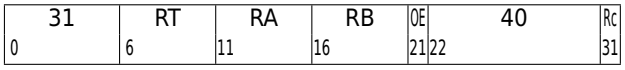
Extended Mnemonics:

Extended Mnemonics for *Add Immediate Carrying and Record*:

Extended mnemonic:		Equivalent to:	
`subic.`	`RT,RA,si`	`addic.`	`RT,RA,-si`

Subtract From XO-form

subf	RT,RA,RB	(OE=0 Rc=0)
subf.	RT,RA,RB	(OE=0 Rc=1)
subfo	RT,RA,RB	(OE=1 Rc=0)
subfo.	RT,RA,RB	(OE=1 Rc=1)




```
RT ← ¬(RA) + (RB) + 1
```

The sum `¬(RA) + (RB) + 1` is placed into register `RT`.

Special Registers Altered:

Register	*Field(s)*	
CR	CR0	(if Rc=1)
XER	SO OV OV32	(if OE=1)

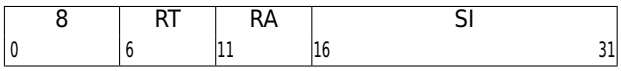
Extended Mnemonics:

Extended Mnemonics for *Subtract From*:

Extended mnemonic:		Equivalent to:	
`sub`	`RT,RB,RA`	`subf`	`RT,RA,RB`
`sub.`	`RT,RB,RA`	`subf.`	`RT,RA,RB`
`subo`	`RT,RB,RA`	`subfo`	`RT,RA,RB`
`subo.`	`RT,RB,RA`	`subfo.`	`RT,RA,RB`

Subtract From Immediate Carrying D-form

subfic RT,RA,SI




```
RT ← ¬(RA) + EXTS(SI) + 1
```

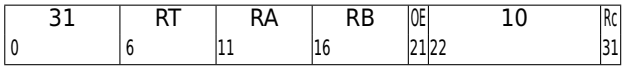
The sum `¬(RA) + SI + 1` is placed into register `RT`.

Special Registers Altered:

Register	*Field(s)*
XER	CA CA32

Add Carrying XO-form

addc	RT,RA,RB	(OE=0 Rc=0)
addc.	RT,RA,RB	(OE=0 Rc=1)
addco	RT,RA,RB	(OE=1 Rc=0)
addco.	RT,RA,RB	(OE=1 Rc=1)




```
RT ← (RA) + (RB)
```

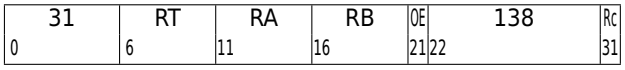
The sum `(RA) + (RB)` is placed into register `RT`.

Special Registers Altered:

Register	*Field(s)*	
CR	CR0	(if Rc=1)
XER	CA CA32	
XER	SO OV OV32	(if OE=1)

Add Extended XO-form

adde	RT,RA,RB	(OE=0 Rc=0)
adde.	RT,RA,RB	(OE=0 Rc=1)
addeo	RT,RA,RB	(OE=1 Rc=0)
addeo.	RT,RA,RB	(OE=1 Rc=1)




```
RT ← (RA) + (RB) + CA
```

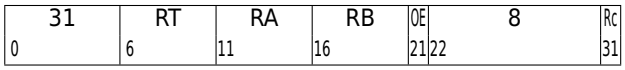
The sum `(RA) + (RB) + CA` is placed into register `RT`.

Special Registers Altered:

Register	*Field(s)*	
CR	CR0	(if Rc=1)
XER	CA CA32	
XER	SO OV OV32	(if OE=1)

Subtract From Carrying XO-form

subfc	RT,RA,RB	(OE=0 Rc=0)
subfc.	RT,RA,RB	(OE=0 Rc=1)
subfco	RT,RA,RB	(OE=1 Rc=0)
subfco.	RT,RA,RB	(OE=1 Rc=1)




```
RT ← ¬(RA) + (RB) + 1
```

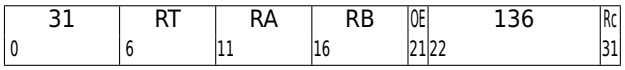
The sum `¬(RA) + (RB) + 1` is placed into register `RT`.

Special Registers Altered:

Register	*Field(s)*	
CR	CR0	(if Rc=1)
XER	CA CA32	
XER	SO OV OV32	(if OE=1)

Subtract From Extended XO-form

subfe	RT,RA,RB	(OE=0 Rc=0)
subfe.	RT,RA,RB	(OE=0 Rc=1)
subfeo	RT,RA,RB	(OE=1 Rc=0)
subfeo.	RT,RA,RB	(OE=1 Rc=1)




```
RT ← ¬(RA) + (RB) + CA
```

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

Special Registers Altered:

Register	*Field(s)*	
CR	CR0	(if Rc=1)
XER	CA CA32	
XER	SO OV OV32	(if OE=1)

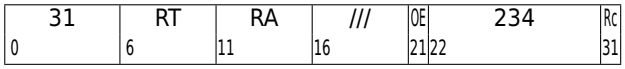
Extended Mnemonics:

Extended Mnemonics for *Subtract From Carrying*:

Extended mnemonic:		**Equivalent to:**	
`subc`	`RT,RB,RA`	`subfc`	`RT,RA,RB`
`subc.`	`RT,RB,RA`	`subfc.`	`RT,RA,RB`
`subco`	`RT,RB,RA`	`subfco`	`RT,RA,RB`
`subco.`	`RT,RB,RA`	`subfco.`	`RT,RA,RB`

Add to Minus One Extended XO-form

addme	RT,RA	(OE=0 Rc=0)
addme.	RT,RA	(OE=0 Rc=1)
addmeo	RT,RA	(OE=1 Rc=0)
addmeo.	RT,RA	(OE=1 Rc=1)




```
RT ← (RA) + CA − 1
```

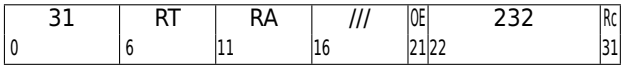
The sum (RA) + CA + $^{64}1$ is placed into register RT.

Special Registers Altered:

Register	*Field(s)*	
CR	CR0	(if Rc=1)
XER	CA CA32	
XER	SO OV OV32	(if OE=1)

Subtract From Minus One Extended XO-form

subfme	RT,RA	(OE=0 Rc=0)
subfme.	RT,RA	(OE=0 Rc=1)
subfmeo	RT,RA	(OE=1 Rc=0)
subfmeo.	RT,RA	(OE=1 Rc=1)




```
RT ← ¬(RA) + CA − 1
```

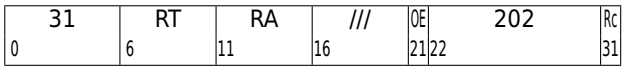
The sum ¬(RA) + CA + $^{64}1$ is placed into register RT.

Special Registers Altered:

Register	*Field(s)*	
CR	CR0	(if Rc=1)
XER	CA CA32	
XER	SO OV OV32	(if OE=1)

Add to Zero Extended XO-form

addze	RT,RA	(OE=0 Rc=0)
addze.	RT,RA	(OE=0 Rc=1)
addzeo	RT,RA	(OE=1 Rc=0)
addzeo.	RT,RA	(OE=1 Rc=1)




```
RT ← (RA) + CA
```

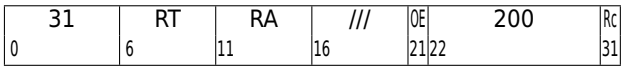
The sum (RA) + CA is placed into register RT.

Special Registers Altered:

Register	*Field(s)*	
CR	CR0	(if Rc=1)
XER	CA CA32	
XER	SO OV OV32	(if OE=1)

Subtract From Zero Extended XO-form

subfze	RT,RA	(OE=0 Rc=0)
subfze.	RT,RA	(OE=0 Rc=1)
subfzeo	RT,RA	(OE=1 Rc=0)
subfzeo.	RT,RA	(OE=1 Rc=1)




```
RT ← ¬(RA) + CA
```

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

Special Registers Altered:

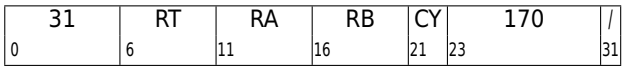
Register	*Field(s)*	
CR	CR0	(if Rc=1)
XER	CA CA32	
XER	SO OV OV32	(if OE=1)

Programming Note

The setting of CA and CA32 by the *Add* and *Subtract From* instructions, including the Extended versions thereof, is mode-dependent. If a sequence of these instructions is used to perform extended-precision addition or subtraction, the same mode should be used throughout the sequence.

Add Extended using alternate carry bit Z23-form

addex RT,RA,RB,CY




```
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:

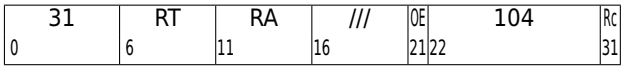
Register	*Field(s)*	
XER	OV OV32	(if CY=0)

Programming Note

An ***addc***-equivalent instruction using OV is not provided. An equivalent capability can be emulated by first initializing OV to 0, then using ***addex***. OV can be initialized to 0 using ***subfo***, subtracting any operand from itself.

Negate XO-form

neg	RT,RA	(OE=0 Rc=0)
neg.	RT,RA	(OE=0 Rc=1)
nego	RT,RA	(OE=1 Rc=0)
nego.	RT,RA	(OE=1 Rc=1)




```
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 is set to 1. If $(RA)_{32:63}$ contain the most negative 32-bit number (0x8000_0000) and OE=1, OV32 is set to 1.

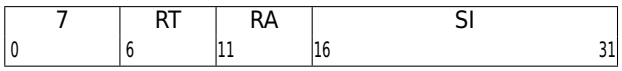
Similarly, if the processor is in 32-bit mode and $(RA)_{32:63}$ contain the most negative 32-bit number (0x8000_0000), the low-order 32 bits of the result contain the most negative 32-bit number and, if OE=1, OV and OV32 are set to 1.

Special Registers Altered:

Register	*Field(s)*	
CR	CR0	(if Rc=1)
XER	SO OV OV32	(if OE=1)

Multiply Low Immediate D-form

mulli	RT,RA,SI	



$prod_{0:127} \leftarrow (RA) \times EXTS(SI)$
$RT \leftarrow prod_{64:127}$

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.

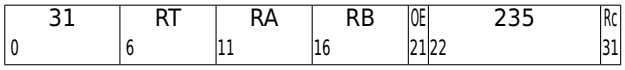
Both operands and the product are interpreted as signed integers.

Special Registers Altered:

None

Multiply Low Word XO-form

mullw	RT,RA,RB	(OE=0 Rc=0)
mullw.	RT,RA,RB	(OE=0 Rc=1)
mullwo	RT,RA,RB	(OE=1 Rc=0)
mullwo.	RT,RA,RB	(OE=1 Rc=1)



$RT \leftarrow (RA)_{32:63} \times (RB)_{32:63}$

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.

If OE=1 then OV and OV32 are set to 1 if the product cannot be represented in 32 bits.

Both operands and the product are interpreted as signed integers.

Special Registers Altered:

Register	*Field(s)*	
CR	CR0	(if Rc=1)
XER	SO OV OV32	(if OE=1)

Programming Note

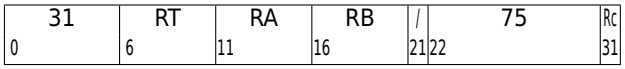
For ***mulli*** and ***mullw***, the low-order 32 bits of the product are the correct 32-bit product for 32-bit mode.

For ***mulli*** and ***mulld***, the low-order 64 bits of the product are independent of whether the operands are regarded as signed or unsigned 64-bit integers.

For ***mulli*** and ***mullw***, the low-order 32 bits of the product are independent of whether the operands are regarded as signed or unsigned 32-bit integers.

Multiply High Word XO-form

mulhw	RT,RA,RB	(Rc=0)
mulhw.	RT,RA,RB	(Rc=1)



$prod_{0:63} \leftarrow (RA)_{32:63} \times (RB)_{32:63}$
$RT_{32:63} \leftarrow prod_{0:31}$
$RT_{0:31} \leftarrow$ undefined

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.

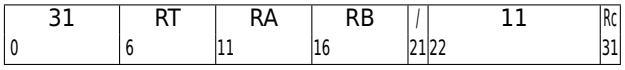
Special Registers Altered:

Register	*Field(s)*	
CR	CR0	(if Rc=1)

(bits 0:2 undefined in 64-bit mode)

Multiply High Word Unsigned XO-form

mulhwu	RT,RA,RB	(Rc=0)
mulhwu.	RT,RA,RB	(Rc=1)



$prod_{0:63} \leftarrow (RA)_{32:63} \times (RB)_{32:63}$
$RT_{32:63} \leftarrow prod_{0:31}$
$RT_{0:31} \leftarrow$ undefined

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.

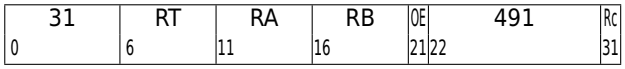
Special Registers Altered:

Register	*Field(s)*	
CR	CR0	(if Rc=1)

(bits 0:2 undefined in 64-bit mode)

Divide Word XO-form

divw	RT,RA,RB	(OE=0 Rc=0)
divw.	RT,RA,RB	(OE=0 Rc=1)
divwo	RT,RA,RB	(OE=1 Rc=0)
divwo.	RT,RA,RB	(OE=1 Rc=1)



$dividend_{0:31} \leftarrow (RA)_{32:63}$
$divisor_{0:31} \leftarrow (RB)_{32:63}$
$RT_{32:63} \leftarrow dividend \div divisor$
$RT_{0:31} \leftarrow$ undefined

The 32-bit dividend is $(RA)_{32:63}$. The 32-bit divisor is $(RB)_{32:63}$. The 32-bit quotient is placed into $RT_{32:63}$. The contents of $RT_{0:31}$ are undefined. The remainder is not supplied as a result.

Both operands and the quotient are interpreted as signed integers. The quotient is the unique signed integer that satisfies

$$dividend = (quotient \times divisor) + r$$

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

If an attempt is made to perform any of the divisions

```
0x8000_0000 ÷ −1
<anything> ÷ 0
```

then the contents of register RT are undefined as are (if Rc=1) the contents of the LT, GT, and EQ bits of CR Field 0. In these cases, if OE=1 then OV and OV32 are set to 1.

Special Registers Altered:

Register	*Field(s)*	
CR	CR0	(if Rc=1)
(bits 0:2 undefined in 64-bit mode)		
XER	SO OV OV32	(if OE=1)

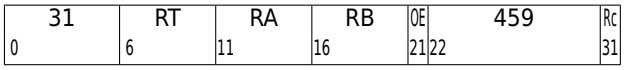
Programming Note

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×divisor
subf    RT,RT,RA    # RT = remainder
```

Divide Word Unsigned XO-form

divwu	RT,RA,RB	(OE=0 Rc=0)
divwu.	RT,RA,RB	(OE=0 Rc=1)
divwuo	RT,RA,RB	(OE=1 Rc=0)
divwuo.	RT,RA,RB	(OE=1 Rc=1)



$dividend_{0:31} \leftarrow (RA)_{32:63}$
$divisor_{0:31} \leftarrow (RB)_{32:63}$
$RT_{32:63} \leftarrow dividend \div divisor$
$RT_{0:31} \leftarrow$ undefined

The 32 bit dividend is $(RA)_{32:63}$. The 32-bit divisor is $(RB)_{32:63}$. The 32-bit quotient is placed into $RT_{32:63}$. The contents of $RT_{0:31}$ are undefined. The remainder is not supplied as a result.

Both operands and the quotient are interpreted as 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 an attempt is made to perform the division

```
<anything> ÷ 0
```

then the contents of register RT are undefined as are (if Rc=1) the contents of the LT, GT, and EQ bits of CR Field 0. In this case, if OE=1 then OV and OV32 are set to 1.

Special Registers Altered:

Register	*Field(s)*	
CR	CR0	(if Rc=1)
(bits 0:2 undefined in 64-bit mode)		
XER	SO OV OV32	(if OE=1)

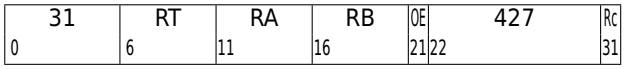
Programming Note

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×divisor
subf    RT,RT,RA    # RT = remainder
```

Divide Word Extended XO-form

divwe	RT,RA,RB	(OE=0 Rc=0)
divwe.	RT,RA,RB	(OE=0 Rc=1)
divweo	RT,RA,RB	(OE=1 Rc=0)
divweo.	RT,RA,RB	(OE=1 Rc=1)



$dividend_{0:63} \leftarrow (RA)_{32:63} \mid\mid {}^{32}0$
$divisor_{0:31} \leftarrow (RB)_{32:63}$
$RT_{32:63} \leftarrow dividend \div divisor$
$RT_{0:31} \leftarrow$ undefined

The 64-bit dividend is $(RA)_{32:63} \mid\mid {}^{32}0$. 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

$$\texttt{<anything>} \div 0$$

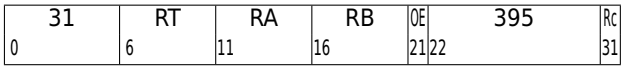
then the contents of register RT are undefined as are (if Rc=1) the contents of the LT, GT, and EQ bits of CR Field 0. In these cases, if OE=1 then OV and OV32 are set to 1.

Special Registers Altered:

Register	*Field(s)*	
CR	CR0	(if Rc=1)
(bits 0:2 undefined in 64-bit mode)		
XER	SO OV OV32	(if OE=1)

Divide Word Extended Unsigned XO-form

divweu	RT,RA,RB	(OE=0 Rc=0)
divweu.	RT,RA,RB	(OE=0 Rc=1)
divweuo	RT,RA,RB	(OE=1 Rc=0)
divweuo.	RT,RA,RB	(OE=1 Rc=1)



$dividend_{0:63} \leftarrow (RA)_{32:63} \mid\mid {}^{32}0$
$divisor_{0:31} \leftarrow (RB)_{32:63}$
$RT_{32:63} \leftarrow dividend \div divisor$
$RT_{0:31} \leftarrow$ undefined

The 64-bit dividend is $(RA)_{32:63} \mid\mid {}^{32}0$. 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

$$\texttt{<anything>} \div 0$$

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:

Register	*Field(s)*	
CR	CR0	(if Rc=1)
(bits 0:2 undefined in 64-bit mode)		
XER	SO OV OV32	(if OE=1)

Programming Note

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:

1. q1 $\leftarrow$ **divweu** Dh, Dv
2. r1 $\leftarrow$ -(q1 $\times$ Dv) # remainder of step 1 divide operation (see Note 1)
3. q2 $\leftarrow$ **divwu** Dl, Dv
4. r2 $\leftarrow$ Dl - (q2 $\times$ Dv) # remainder of step 2 divide operation
5. Q $\leftarrow$ q1 + q2
6. R $\leftarrow$ r1 + r2
7. if (R $<$ r2) | (R $\geq$ Dv) then # (see Note 2)
   Q $\leftarrow$ Q + 1 # increment quotient
   R $\leftarrow$ R - Dv # decrement remainder

Assembler Code:

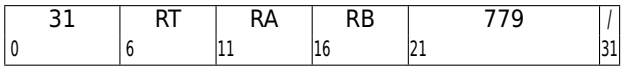
```
# Dh in r4, Dl in r5
# Dv in r6
divweu  r3,r4,r6        # q1
divwu   r7,r5,r6        # q2
mullw   r8,r3,r6        # -r1 = q1 * Dv
mullw   r0,r7,r6        # q2 * Dv
subf    r10,r0,r5       # r2 = Dl - (q2 * Dv)
add     r3,r3,r7        # Q = q1 + q2
subf    r4,r8,r10       # R = r1 + r2
cmplw   r4,r10          # R < r2 ?
blt     *+12            # must adjust Q and R if yes
cmplw   r4,r6           # R ≥ 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 || `$^{32}0$ `- (q1 × Dv)`. Because the remainder must be less than `Dv` and `Dv` $< 2^{32}$, the remainder is representable in 32 bits. Because the low-order 32 bits of `Dh || `$^{32}0$ are 0s, the remainder is therefore equal to the low-order 32 bits of `-(q1 × Dv)`. Thus assigning `-(q1 × 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`.
3. For additional information see the book *Hacker's Delight* by Henry S. Warren, Jr. []

Modulo Signed Word X-form

modsw RT,RA,RB




```
dividend0:31  ← (RA)32:63
divisor0:31   ← (RB)32:63
RT32:63       ← dividend % divisor
RT0:31        ← undefined
```

The 32-bit dividend is $(RA)_{32:63}$. The 32-bit divisor is $(RB)_{32:63}$. The 32-bit remainder of the dividend divided by the divisor is placed into $RT_{32:63}$. The contents of $RT_{0:31}$ are undefined. The quotient is not supplied as a result.

Both operands and the remainder are interpreted as signed integers. The remainder is the unique signed integer that satisfies

```
remainder = dividend − (quotient ×
divisor)
```

where $0 \leq remainder < |divisor|$ if the dividend is nonnegative, and $-|divisor| < remainder \leq 0$ if the dividend is negative.

If an attempt is made to perform any of the divisions

```
0x8000_0000 % −1
<anything> % 0
```

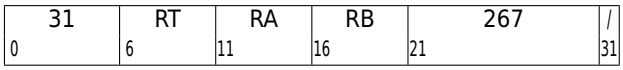
then the contents of register RT are undefined.

Special Registers Altered:

None

Modulo Unsigned Word X-form

moduw RT,RA,RB




```
dividend0:31  ← (RA)32:63
divisor0:31   ← (RB)32:63
RT32:63       ← dividend % divisor
RT0:31        ← undefined
```

The 32-bit dividend is $(RA)_{32:63}$. The 32-bit divisor is $(RB)_{32:63}$. The 32-bit remainder of the dividend divided by the divisor is placed into $RT_{32:63}$. The contents of $RT_{0:31}$ are undefined. The quotient is not supplied as a result.

Both operands and the remainder are interpreted as unsigned integers. The remainder is the unique signed integer that satisfies

```
remainder = dividend − (quotient ×
divisor)
```

where $0 \leq remainder < divisor$.

If an attempt is made to perform any of the divisions

```
<anything> % 0
```

then the contents of register RT are undefined.

Special Registers Altered:

None

Deliver A Random Number X-form

darn RT,L

31	RT	///	L	///	755	/
0	6	11	14	16	21	31

```
RT ← random(L)
```

A random number is placed into register RT in a format selected by L as shown in the following table. The value 0xFFFFFFFF_FFFFFFFF indicates an error condition. For L=0, the random number range is 0:0xFFFFFFFF. For L=1 and L=2, the random number range is 0:0xFFFFFFFF_FFFFFFFE.

L	Format
0	$^{32}0$ || $CRN_{0:31}$
1	$CRN_{0:63}$
2	$RRN_{0:63}$
3	reserved
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 ***darn*** and interpret the value 0xFFFFFFFF to indicate an error condition. The fact that the error condition includes the valid value 0x00000000_FFFFFFFF together with the true error value 0xFFFFFFFF_FFFFFFFF is not a problem.

Programming Note

When the error value is obtained, software is expected to repeat the operation. If a non-error value has not been obtained after several attempts, a software random number generation method should be used. The recommended number of attempts may be implementation specific. In the absence of other guidance, ten attempts should be adequate.

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.

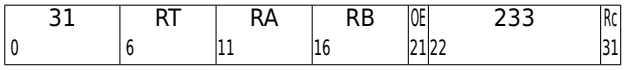
Engineering Note

To preserve the ability for a function to be assigned for L=3 and for that to be emulated on implementations that comply with versions of the architecture in which 3 is a reserved value for L, ***darn*** with L=3 should cause a Hypervisor Emulation Assistance interrupt.

3.3.9.1 64-bit Fixed-Point Arithmetic Instructions

Multiply Low Doubleword XO-form

mulld	RT,RA,RB	(OE=0 Rc=0)
mulld.	RT,RA,RB	(OE=0 Rc=1)
mulldo	RT,RA,RB	(OE=1 Rc=0)
mulldo.	RT,RA,RB	(OE=1 Rc=1)



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

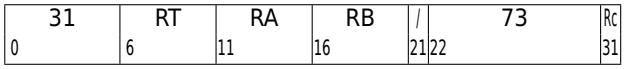
Register	*Field(s)*	
CR	CR0	(if Rc=1)
XER	SO OV OV32	(if OE=1)

Programming Note

The XO-form *Multiply* instructions may execute faster on some implementations if RB contains the operand having the smaller absolute value.

Multiply High Doubleword XO-form

mulhd	RT,RA,RB	(Rc=0)
mulhd.	RT,RA,RB	(Rc=1)



$prod_{0:127} \leftarrow$ (RA) $\times$ (RB)
RT $\leftarrow$ $prod_{0:63}$

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.

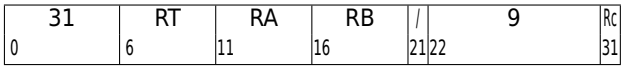
Both operands and the product are interpreted as signed integers.

Special Registers Altered:

Register	*Field(s)*	
CR	CR0	(if Rc=1)

Multiply High Doubleword Unsigned XO-form

mulhdu	RT,RA,RB	(Rc=0)
mulhdu.	RT,RA,RB	(Rc=1)



$prod_{0:127} \leftarrow$ (RA) $\times$ (RB)
RT $\leftarrow$ $prod_{0:63}$

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:

Register	*Field(s)*	
CR	CR0	(if Rc=1)

Multiply-Add High Doubleword VA-form

maddhd RT,RA,RB,RC

4	RT	RA	RB	RC	48
0	6	11	16	21	26 31

```
prod0:127 ← (RA) × (RB)
sum0:127  ← prod + EXTS(RC)
RT ← sum0:63
```

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

maddhdu RT,RA,RB,RC

4	RT	RA	RB	RC	49
0	6	11	16	21	26 31

```
prod0:127 ← (RA) × (RB)
sum0:127  ← prod + EXTZ(RC)
RT ← sum0:63
```

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

maddld RT,RA,RB,RC

4	RT	RA	RB	RC	51
0	6	11	16	21	26 31

```
prod0:127 ← (RA) × (RB)
sum0:127  ← prod + EXTS(RC)
RT ← sum64:127
```

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.

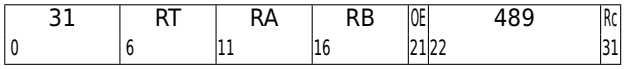
All three operands and the result are interpreted as signed integers.

Special Registers Altered:

None

Divide Doubleword XO-form

divd	RT,RA,RB	(OE=0 Rc=0)
divd.	RT,RA,RB	(OE=0 Rc=1)
divdo	RT,RA,RB	(OE=1 Rc=0)
divdo.	RT,RA,RB	(OE=1 Rc=1)



$dividend_{0:63} \leftarrow$ (RA)
$divisor_{0:63} \leftarrow$ (RB)
RT $\leftarrow$ dividend $\div$ divisor

The 64-bit dividend is (RA). The 64-bit divisor is (RB). The 64-bit quotient is placed into register RT. The remainder is not supplied as a result.

Both operands and the quotient are interpreted as signed integers. The quotient is the unique signed integer that satisfies

$$dividend = (quotient \times divisor) + r$$

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

If an attempt is made to perform any of the divisions

0x8000_0000_0000_0000 $\div$ −1
<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:

Register	*Field(s)*	
CR	CR0	(if Rc=1)
XER	SO OV OV32	(if OE=1)

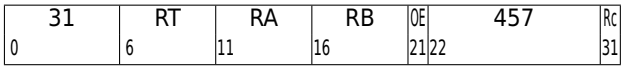
Programming Note

The 64-bit signed remainder of dividing (RA) by (RB) can be computed as follows, except in the case that (RA) = -2^{63} and (RB) = −1.

```
divd   RT,RA,RB  # RT = quotient
mulld  RT,RT,RB  # RT = quotient×divisor
subf   RT,RT,RA  # RT = remainder
```

Divide Doubleword Unsigned XO-form

divdu	RT,RA,RB	(OE=0 Rc=0)
divdu.	RT,RA,RB	(OE=0 Rc=1)
divduo	RT,RA,RB	(OE=1 Rc=0)
divduo.	RT,RA,RB	(OE=1 Rc=1)



$dividend_{0:63} \leftarrow$ (RA)
$divisor_{0:63} \leftarrow$ (RB)
RT $\leftarrow$ dividend $\div$ divisor

The 64-bit dividend is (RA). The 64-bit divisor is (RB). The 64-bit quotient is placed into register RT. The remainder is not supplied as a result.

Both operands and the quotient are interpreted as 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 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 this case, if OE=1 then OV and OV32 are set to 1.

Special Registers Altered:

Register	*Field(s)*	
CR	CR0	(if Rc=1)
XER	SO OV OV32	(if OE=1)

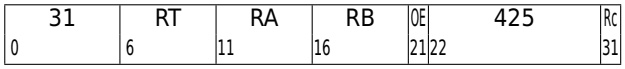
Programming Note

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

```
divdu  RT,RA,RB  # RT = quotient
mulld  RT,RT,RB  # RT = quotient×divisor
subf   RT,RT,RA  # RT = remainder
```

Divide Doubleword Extended XO-form

divde	RT,RA,RB	(OE=0 Rc=0)
divde.	RT,RA,RB	(OE=0 Rc=1)
divdeo	RT,RA,RB	(OE=1 Rc=0)
divdeo.	RT,RA,RB	(OE=1 Rc=1)




```
dividend0:127 ← (RA) || 64 0
divisor0:63 ← (RB)
RT ← dividend ÷ divisor
```

The 128-bit dividend is `(RA) || 64 0`. 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

dividend = (*quotient* × *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 64 bits, or if an attempt is made to perform the division

```
<anything> ÷ 0
```

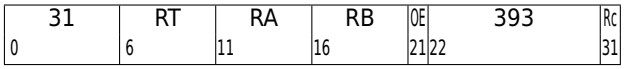
then the contents of register `RT` are undefined as are (if Rc=1) the contents of the `LT`, `GT`, and `EQ` bits of CR Field 0. In these cases, if `OE=1` then `OV` and `OV32` are set to 1.

Special Registers Altered:

Register	*Field(s)*	
CR	CR0	(if Rc=1)
XER	SO OV OV32	(if OE=1)

Divide Doubleword Extended Unsigned XO-form

divdeu	RT,RA,RB	(OE=0 Rc=0)
divdeu.	RT,RA,RB	(OE=0 Rc=1)
divdeuo	RT,RA,RB	(OE=1 Rc=0)
divdeuo.	RT,RA,RB	(OE=1 Rc=1)




```
dividend0:127 ← (RA) || 64 0
divisor0:63 ← (RB)
RT ← dividend ÷ divisor
```

The 128-bit dividend is `(RA) || 64 0`. 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

dividend = (*quotient* × *divisor*) + *r*

where $0 \leq r < divisor$.

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

```
<anything> ÷ 0
```

then the contents of register `RT` are undefined as are (if `Rc=1`) the contents of the `LT`, `GT`, and `EQ` bits of CR Field 0. In these cases, if `OE=1` then `OV` and `OV32` are set to 1.

Special Registers Altered:

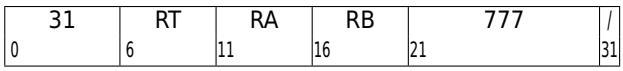
Register	*Field(s)*	
CR	CR0	(if Rc=1)
XER	SO OV OV32	(if OE=1)

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 ***divweu*** instruction description: ***divd***[***e***]***u*** would be used instead of ***divw***[***e***]***u*** (and ***cmpld*** instead of ***cmplw***, etc.).

Modulo Signed Doubleword X-form

modsd RT,RA,RB




```
dividend ← (RA)
divisor  ← (RB)
RT       ← dividend % divisor
```

The 64-bit dividend is (RA). The 64-bit divisor is (RB). The 64-bit remainder of the dividend divided by the divisor is placed into register RT. The quotient is not supplied as a result.

Both operands and the remainder are interpreted as signed integers. The remainder is the unique signed integer that satisfies

$$\text{remainder} = \text{dividend} - (\text{quotient} \times \text{divisor})$$

where $0 \leq \text{remainder} < |\text{divisor}|$ if the dividend is nonnegative, and $-|\text{divisor}| < \text{remainder} \leq 0$ if the dividend is negative.

If an attempt is made to perform any of the divisions

```
<anything> % 0
0x8000_0000_0000_0000 % -1
```

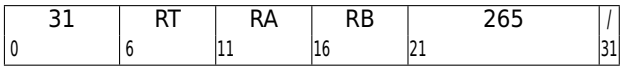
then the contents of register RT are undefined.

Special Registers Altered:

None

Modulo Unsigned Doubleword X-form

modud RT,RA,RB




```
dividend ← (RA)
divisor  ← (RB)
RT       ← dividend % divisor
```

The 64-bit dividend is (RA). The 64-bit divisor is (RB). The 64-bit remainder of the dividend divided by the divisor is placed into register RT. The quotient is not supplied as a result.

Both operands and the remainder are interpreted as unsigned integers. The remainder is the unique signed integer that satisfies

$$\text{remainder} = \text{dividend} - (\text{quotient} \times \text{divisor})$$

where $0 \leq \text{remainder} < \text{divisor}$.

If an attempt is made to perform any of the divisions

```
<anything> % 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 ***cmpi*** and ***cmp***, and unsigned for ***cmpli*** and ***cmpl***.

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

L	Operand length
0	32-bit operands
1	64-bit operands

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.

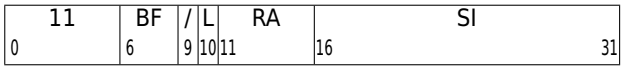
Bit	Name	Description
0	LT	(RA) $<$ SI or (RB) (signed comparison) (RA) $<^{u}$ UI or (RB) (unsigned comparison)
1	GT	(RA) $>$ SI or (RB) (signed comparison) (RA) $>^{u}$ UI or (RB) (unsigned comparison)
2	EQ	(RA) = SI, UI, or (RB)
3	SO	Summary Overflow from the XER

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 D-form

cmpi BF,L,RA,SI




```
if L = 0 then a ← EXTS((RA)32:63)
         else a ← (RA)
if      a < EXTS(SI) then c ← 0b100
else if a > EXTS(SI) then c ← 0b010
else                      c ← 0b001
CR4×BF+32:4×BF+35 ← c || XERSO
```

The contents of register RA ($(RA)_{32:63}$ sign-extended to 64 bits if L=0) are compared with the sign-extended value of the SI field, treating the operands as signed integers. The result of the comparison is placed into CR field BF.

Special Registers Altered:

Register	*Field(s)*
CR	field BF

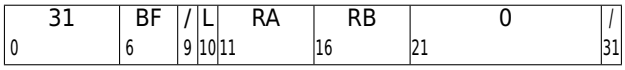
Extended Mnemonics:

Extended Mnemonics for *Compare Immediate*:

Extended mnemonic:		Equivalent to:	
cmpdi	[BF=0,]RA,SI	cmpi	BF,1,RA,SI
cmpwi	[BF=0,]RA,SI	cmpi	BF,0,RA,SI

Compare X-form

cmp BF,L,RA,RB




```
if L = 0 then a ← EXTS((RA)32:63)
              b ← EXTS((RB)32:63)
         else a ← (RA)
              b ← (RB)
if      a < b then c ← 0b100
else if a > b then c ← 0b010
else               c ← 0b001
CR4×BF+32:4×BF+35 ← c || 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:

Register	*Field(s)*
CR	field BF

Extended Mnemonics:

Extended Mnemonics for *Compare*:

Extended mnemonic:		Equivalent to:	
cmpd	[BF=0,]RA,RB	cmp	BF,1,RA,RB
cmpw	[BF=0,]RA,RB	cmp	BF,0,RA,RB

Compare Logical Immediate D-form

cmpli BF,L,RA,UI

10	BF	/	L	RA	UI
0	6	9	10	11	16 31

```
if L = 0 then a ← 32 0 || (RA)32:63
         else a ← (RA)
if      a <u (48 0 || UI) then c ← 0b100
else if a >u (48 0 || UI) then c ← 0b010
else                           c ← 0b001
CR4×BF+32:4×BF+35 ← c || XERSO
```

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:

Register	*Field(s)*
CR	field BF

Extended Mnemonics:

Extended Mnemonics for *Compare Logical Immediate*:

Extended mnemonic:		Equivalent to:	
cmpldi	[BF=0,]RA,UI	cmpli	BF,1,RA,UI
cmplwi	[BF=0,]RA,UI	cmpli	BF,0,RA,UI

Compare Logical X-form

cmpl BF,L,RA,RB

31	BF	/	L	RA	RB	32	/
0	6	9	10	11	16	21	31

```
if L = 0 then a ← 32 0 || (RA)32:63
              b ← 32 0 || (RB)32:63
         else a ← (RA)
              b ← (RB)
if      a <u b then c ← 0b100
else if a >u b then c ← 0b010
else                c ← 0b001
CR4×BF+32:4×BF+35 ← c || 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 unsigned integers. The result of the comparison is placed into CR field BF.

Special Registers Altered:

Register	*Field(s)*
CR	field BF

Extended Mnemonics:

Extended Mnemonics for *Compare Logical*:

Extended mnemonic:		Equivalent to:	
cmpld	[BF=0,]RA,RB	cmpl	BF,1,RA,RB
cmplw	[BF=0,]RA,RB	cmpl	BF,0,RA,RB

3.3.10.1 Character-Type Compare Instructions

Compare Ranged Byte X-form

cmprb BF,L,RA,RB

31	BF	/	L	RA	RB	192	/
0	6	9	10	11	16	21	31

```
src1    ← EXTZ((RA)56:63)

src21hi ← EXTZ((RB)32:39)
src21lo ← EXTZ((RB)40:47)
src22hi ← EXTZ((RB)48:55)
src22lo ← EXTZ((RB)56:63)

if L=0 then
   in_range ← (src22lo ≤ src1) &
              (src1 ≤ src22hi)
else
   in_range ← ((src21lo ≤ src1) &
               (src1 ≤ src21hi)) |
              ((src22lo ≤ src1) &
               (src1 ≤ src22hi))

CR4×BF+32:4×BF+35 ← 0b0 || in_range || 0b00
```

Let src1 be the unsigned integer value in bits 56:63 of register RA.

Let src21hi be the unsigned integer value in bits 32:39 of register RB.

Let src21lo be the unsigned integer value in bits 40:47 of register RB.

Let src22hi be the unsigned integer value in bits 48:55 of register RB.

Let src22lo be the unsigned integer value in bits 56:63 of register RB.

Let x be considered “in range” of y:z if the value x is greater than or equal to the value y and the value x is less than or equal to the value z.

When L=0, the value in_range is set to 1 if src1 is in range of src22lo:src22hi. Otherwise, the value in_range is set to 0.

When L=1, the value in_range is set to 1 if either src1 is in range of src21lo:src21hi, or src1 is in range of src22lo:src22hi. Otherwise, the value in_range is set to 0.

CR field BF is set to the value 0b0 concatenated with in_range concatenated with 0b00.

Special Registers Altered:

Register	*Field(s)*
CR	field BF

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
```

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

```
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
```

Compare Equal Byte X-form

cmpeqb BF,RA,RB

31	BF	//	RA	RB	224	/
0	6	9	11	16	21	31

```
src1 ← GPR[RA].bit[56:63]

match ← (src1 = (RB)00:07) |
         (src1 = (RB)08:15) |
         (src1 = (RB)16:23) |
         (src1 = (RB)24:31) |
         (src1 = (RB)32:39) |
         (src1 = (RB)40:47) |
         (src1 = (RB)48:55) |
         (src1 = (RB)56:63)

CR4×BF+32:4×BF+35 ← 0b0 || match || 0b00
```

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:

Register	*Field(s)*
CR	field BF

Programming Note

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

A function such as `isspace()` can be implemented by loading the 6 byte codes corresponding to characters considered as whitespace (HT, LF, VT, FF, CR, and SP) and using the ***cmpeqb*** to compare the subject character to those 6 values to determine if any match occurs.

```
ldx     rSPC,WS_CHARS       ; rSPC = 0x0909_090A_0B0C_0D20
                            ; load rSPC with all 6 ASCII values corresponding to white spaces
cmpeqb  2,cr1,rCHAR,rSPC    ; perform match compare on character in rCHAR with byte values in rSPC
```

In this case, the byte code for HT (0x09) was replicated to fill 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.

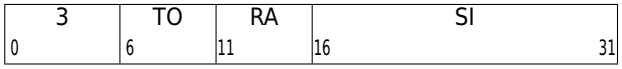
TO Bit	ANDed with Condition
0	Less Than, using signed comparison
1	Greater Than, using signed comparison
2	Equal
3	Less Than, using unsigned comparison
4	Greater Than, using unsigned comparison

Extended mnemonics for traps

A set of extended mnemonics is provided so that traps can be coded with the condition as part of the mnemonic rather than as a numeric operand. Some of these are shown as examples with the Trap instructions. See Appendix C for additional extended mnemonics.

Trap Word Immediate D-form

twi TO,RA,SI




```
a ← EXTS((RA)32:63)
if (a < EXTS(SI)) & TO0  then TRAP
if (a > EXTS(SI)) & TO1  then TRAP
if (a = EXTS(SI)) & TO2  then TRAP
if (a <u EXTS(SI)) & TO3 then TRAP
if (a >u EXTS(SI)) & TO4 then TRAP
```

The contents of $RA_{32:63}$ 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

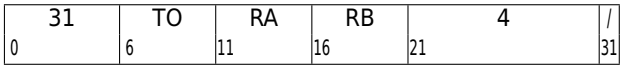
Extended Mnemonics:

Examples of extended mnemonics for *Trap Word Immediate*:

Extended mnemonic:		Equivalent to:	
twgti	RA,SI	twi	8,RA,SI
twllei	RA,SI	twi	6,RA,SI

Trap Word X-form

tw TO,RA,RB




```
a ← EXTS((RA)32:63)
b ← EXTS((RB)32:63)
if (a < b) & TO0 then TRAP
if (a > b) & TO1 then TRAP
if (a = b) & TO2 then TRAP
if (a <u b) & TO3 then TRAP
if (a >u b) & TO4 then TRAP
```

The contents of $RA_{32:63}$ are compared with the contents of $RB_{32:63}$. 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:

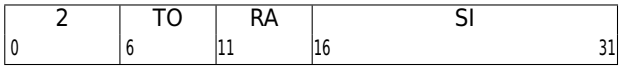
Examples of extended mnemonics for *Trap Word*:

Extended mnemonic:		Equivalent to:	
tweq	RA,RB	tw	4,RA,RB
twlge	RA,RB	tw	5,RA,RB
trap		tw	31,0,0

3.3.11.1 64-bit Fixed-Point Trap Instructions

Trap Doubleword Immediate D-form

tdi TO,RA,SI




```
a ← (RA)
b ← EXTS(SI)
if (a < b) & TO0 then TRAP
if (a > b) & TO1 then TRAP
if (a = b) & TO2 then TRAP
if (a <u b) & TO3 then TRAP
if (a >u b) & TO4 then TRAP
```

The contents of 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

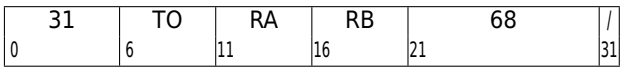
Extended Mnemonics:

Examples of extended mnemonics for *Trap Doubleword Immediate*:

Extended mnemonic:		Equivalent to:	
tdlti	RA,SI	tdi	16,RA,SI
tdnei	RA,SI	tdi	24,RA,SI

Trap Doubleword X-form

td TO,RA,RB




```
a ← (RA)
b ← (RB)
if (a < b) & TO0 then TRAP
if (a > b) & TO1 then TRAP
if (a = b) & TO2 then TRAP
if (a <u b) & TO3 then TRAP
if (a >u b) & TO4 then TRAP
```

The contents of 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 mnemonic:		Equivalent to:	
tdge	RA,RB	td	12,RA,RB
tdlnl	RA,RB	td	5,RA,RB

3.3.12 Fixed-Point Select

Integer Select A-form

isel RT,RA,RB,BC

31	RT	RA	RB	BC	15	/
0	6	11	16	21	26	31

```
if RA=0 then a ← 0 else a ← (RA)
if CR_BC+32=1 then
   RT ← a
else
   RT ← (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:

Extended Mnemonics for *Integer Select*:

Extended mnemonic:		Equivalent to:	
isellt	RT,RA,RB	isel	RT,RA,RB,0
iselgt	RT,RA,RB	isel	RT,RA,RB,1
iseleq	RT,RA,RB	isel	RT,RA,RB,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 ***andi.*** and ***andis.***, set the first three bits of CR Field 0 as described in Section 3.3.8, "Other Fixed-Point Instructions" on page 72. 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 *OR Immediate* instruction. The second type is the executed form, which is intended to consume the same amount of the processor's execution resources as if it were not a no-op. This form is based on the *XOR Immediate* instruction. (There are also no-ops that have other uses, such as affecting program priority, for which extended mnemonics have not been defined.)

Extended mnemonics are provided that use the *OR* and *NOR* instructions to copy the contents of one register to another, with and without complementing. These are shown as examples with the two instructions.

See Appendix C, "Assembler Extended Mnemonics" on page 954 for additional extended mnemonics.

Programming Note

Warning: Some forms of no-op may have side effects such as affecting program priority. Programmers should use the preferred no-op unless the side effects of some other form of no-op are intended.

AND Immediate D-form

andi. RA,RS,UI

28	RS	RA	UI
0	6	11	16 31

```
RA ← (RS) & (48 0 || UI)
```

The contents of register RS are ANDed with $^{48}0$ || UI and the result is placed into register RA.

Special Registers Altered:

Register	*Field(s)*
CR	CR0

AND Immediate Shifted D-form

andis. RA,RS,UI

29	RS	RA	UI
0	6	11	16 31

```
RA ← (RS) & (32 0 || UI || 16 0)
```

The contents of register RS are ANDed with $^{32}0$ || UI || $^{16}0$ and the result is placed into register RA.

Special Registers Altered:

Register	*Field(s)*
CR	CR0

OR Immediate D-form

ori RA,RS,UI

24	RS	RA	UI
0	6	11	16 31

RA ← (RS) | (^{48}0 || UI)

The contents of register RS are ORed with ^{48}0 || UI and the result is placed into register RA.

The preferred “no-op” (an instruction that does nothing) is:

```
ori   0,0,0
```

Some other forms of *ori Rx,Rx,0* provide special functions; see Section 9.2.1 of Book III.

Special Registers Altered:

None

Extended Mnemonics:

Extended Mnemonics for *OR Immediate*:

Extended mnemonic:	Equivalent to:
nop	ori 0,0,0

Engineering Note

It is desirable for implementations to make the preferred form of no-op execute quickly, since this form should be used by compilers.

OR Immediate Shifted D-form

oris RA,RS,UI

25	RS	RA	UI
0	6	11	16 31

RA ← (RS) | (^{32}0 || UI || ^{16}0)

The contents of register RS are ORed with ^{32}0 || UI || ^{16}0 and the result is placed into register RA.

Special Registers Altered:

None

XOR Immediate D-form

xori RA,RS,UI

26	RS	RA	UI
0	6	11	16 31

RA ← (RS) XOR (^{48}0 || UI)

The contents of register RS are XORed with ^{48}0 || UI and the result is placed into register RA.

The executed form of a “no-op” (an instruction that does nothing, but consumes execution resources nevertheless) is:

```
xori   0,0,0
```

Special Registers Altered:

None

Extended Mnemonics:

Extended Mnemonics for *XOR Immediate*:

Extended mnemonic:	Equivalent to:
xnop	xori 0,0,0

Programming Note

The executed form of no-op should be used only when the intent is to alter the timing of a program.

Engineering Note

Implementations must exclude the executed form of no-op from optimizations related to no-ops. These optimizations include, but are not limited to, removal of the instruction from the execution stream, and ignoring register dependencies between subsequent instances of the instruction.

XOR Immediate Shifted D-form

xoris RA,RS,UI

27	RS	RA	UI
0	6	11	16 31

RA ← (RS) XOR (^{32}0 || UI || ^{16}0)

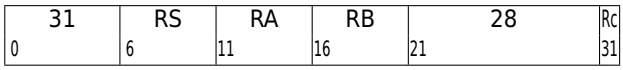
The contents of register RS are XORed with ^{32}0 || UI || ^{16}0 and the result is placed into register RA.

Special Registers Altered:

None

AND X-form

and	RA,RS,RB	(Rc=0)
and.	RA,RS,RB	(Rc=1)




```
RA ← (RS) & (RB)
```

The contents of register RS are ANDed with the contents of register RB and the result is placed into register RA.

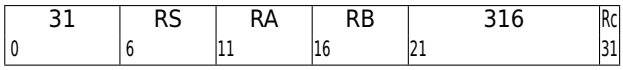
Some forms of ***and*** Rx, Rx, Rx provide special functions; see Section 11.3 of Book III.

Special Registers Altered:

Register	*Field(s)*	
CR	CR0	(if Rc=1)

XOR X-form

xor	RA,RS,RB	(Rc=0)
xor.	RA,RS,RB	(Rc=1)




```
RA ← (RS) ⊕ (RB)
```

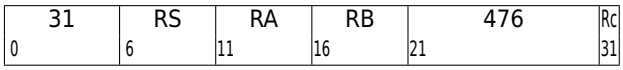
The contents of register RS are XORed with the contents of register RB and the result is placed into register RA.

Special Registers Altered:

Register	*Field(s)*	
CR	CR0	(if Rc=1)

NAND X-form

nand	RA,RS,RB	(Rc=0)
nand.	RA,RS,RB	(Rc=1)




```
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:

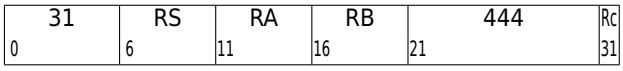
Register	*Field(s)*	
CR	CR0	(if Rc=1)

Programming Note

nand or ***nor*** with RS=RB can be used to obtain the one's complement.

OR X-form

or	RA,RS,RB	(Rc=0)
or.	RA,RS,RB	(Rc=1)




```
RA ← (RS) | (RB)
```

The contents of register RS are ORed with the contents of register RB and the result is placed into register RA.

Some forms of *or Rx,Rx,Rx* provide special functions; see Section 3.2 of Book II and Section 4.3.3 of Book II.

Special Registers Altered:

Register	*Field(s)*	
CR	CR0	(if Rc=1)

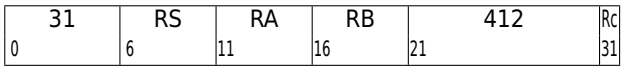
Extended Mnemonics:

Extended Mnemonics for *OR*:

Extended mnemonic:		**Equivalent to:**	
mr	RA,RS	or	RA,RS,RS
mr.	RA,RS	or.	RA,RS,RS

OR with Complement X-form

orc	RA,RS,RB	(Rc=0)
orc.	RA,RS,RB	(Rc=1)




```
RA ← (RS) | ¬(RB)
```

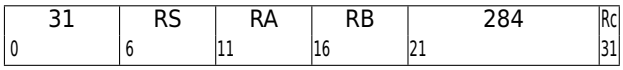
The contents of register RS are ORed with the complement of the contents of register RB and the result is placed into register RA.

Special Registers Altered:

Register	*Field(s)*	
CR	CR0	(if Rc=1)

Equivalent X-form

eqv	RA,RS,RB	(Rc=0)
eqv.	RA,RS,RB	(Rc=1)




```
RA ← (RS) ≡ (RB)
```

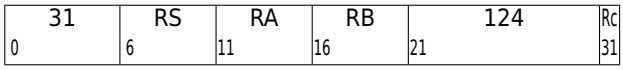
The contents of register RS are XORed with the contents of register RB and the complemented result is placed into register RA.

Special Registers Altered:

Register	*Field(s)*	
CR	CR0	(if Rc=1)

NOR X-form

nor	RA,RS,RB	(Rc=0)
nor.	RA,RS,RB	(Rc=1)




```
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:

Register	*Field(s)*	
CR	CR0	(if Rc=1)

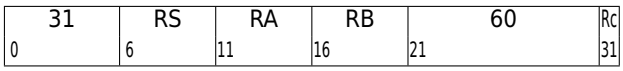
Extended Mnemonics:

Extended Mnemonics for *NOR*:

Extended mnemonic:		**Equivalent to:**	
not	RA,rs	nor	RA,rs,rs
not.	RA,rs	nor.	RA,rs,rs

AND with Complement X-form

andc	RA,RS,RB	(Rc=0)
andc.	RA,RS,RB	(Rc=1)




```
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:

Register	*Field(s)*	
CR	CR0	(if Rc=1)

Extend Sign Byte X-form

extsb	RA,RS	(Rc=0)
extsb.	RA,RS	(Rc=1)

31	RS	RA	///	954	Rc
0	6	11	16	21	31

```
s ← (RS)56
RA56:63 ← (RS)56:63
RA0:55 ← 56s
```

$(RS)_{56:63}$ are placed into $RA_{56:63}$. $RA_{0:55}$ are filled with a copy of $(RS)_{56}$.

Special Registers Altered:

Register	*Field(s)*	
CR	CR0	(if Rc=1)

Extend Sign Halfword X-form

extsh	RA,RS	(Rc=0)
extsh.	RA,RS	(Rc=1)

31	RS	RA	///	922	Rc
0	6	11	16	21	31

```
s ← (RS)48
RA48:63 ← (RS)48:63
RA0:47 ← 48s
```

$(RS)_{48:63}$ are placed into $RA_{48:63}$. $RA_{0:47}$ are filled with a copy of $(RS)_{48}$.

Special Registers Altered:

Register	*Field(s)*	
CR	CR0	(if Rc=1)

Compare Bytes X-form

cmpb	RA,RS,RB

31	RS	RA	RB	508	Rc
0	6	11	16	21	31

```
do n = 0 to 7
   if RS8×n:8×n+7 = (RB)8×n:8×n+7 then
      RA8×n:8×n+7 ← 81
   else
      RA8×n:8×n+7 ← 80
```

Each byte of the contents of register RS is compared to each corresponding byte of the contents in register RB. If they are equal, the corresponding byte in RA is set to 0xFF. Otherwise the corresponding byte in RA is set to 0x00.

Special Registers Altered:

None

Count Leading Zeros Word X-form

cntlzw	RA,RS	(Rc=0)
cntlzw.	RA,RS	(Rc=1)

31	RS	RA	///	26	Rc
0	6	11	16	21	31

```
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:

Register	*Field(s)*	
CR	CR0	(if Rc=1)

Programming Note

For both *Count Leading Zeros* instructions, if Rc=1 then LT is set to 0 in CR Field 0.

Count Trailing Zeros Word X-form

cnttzw	RA,RS	(Rc=0)
cnttzw.	RA,RS	(Rc=1)

31	RS	RA	///	538	Rc
0	6	11	16	21	31

```
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.

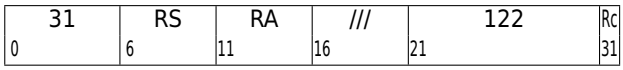
If Rc is equal to 1, CR field 0 is set to reflect the result.

Special Registers Altered:

Register	*Field(s)*	
CR	CR0	(if Rc=1)

Population Count Bytes X-form

popcntb RA,RS




```
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
```

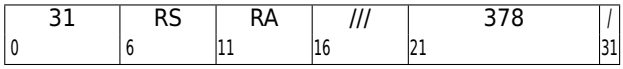
A count of the number of one bits in each byte of register RS is placed into the corresponding byte of register RA. This number ranges from 0 to 8, inclusive.

Special Registers Altered:

None

Population Count Words X-form

popcntw RA,RS




```
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
```

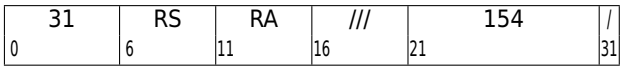
A count of the number of one bits in each word of register RS is placed into the corresponding word of register RA. This number ranges from 0 to 32, inclusive.

Special Registers Altered:

None

Parity Word X-form

prtyw RA,RS




```
s ← 0
t ← 0
do i = 0 to 3
   s ← s ⊕ (RS)i×8+7
do i = 4 to 7
   t ← t ⊕ (RS)i×8+7
RA0:31 ← 310 || s
RA32:63 ← 310 || 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.

```
popcntb  RA, RS
prtyw    RA, RA
```

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

```
popcntb  RA, RS
prtyd    RA, RA
```

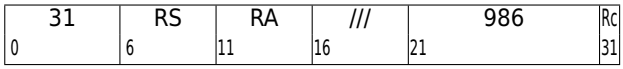
Architecture Note

The *Parity* instructions do not have a record form because of the anticipated difficulty in supporting such forms in a high-frequency pipelined implementation.

3.3.13.1 64-bit Fixed-Point Logical Instructions

Extend Sign Word X-form

extsw	RA,RS	(Rc=0)
extsw.	RA,RS	(Rc=1)




```
s ← (RS)32
RA32:63 ← (RS)32:63
RA0:31 ← 32s
```

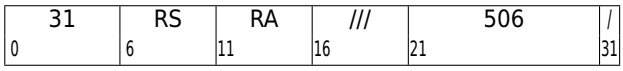
$(RS)_{32:63}$ are placed into $RA_{32:63}$. $RA_{0:31}$ are filled with a copy of $(RS)_{32}$.

Special Registers Altered:

Register	*Field(s)*	
CR	CR0	(if Rc=1)

Population Count Doubleword X-form

popcntd RA,RS




```
n ← 0
do i = 0 to 63
   if (RS)i = 1 then
   n ← n+1
RA ← n
```

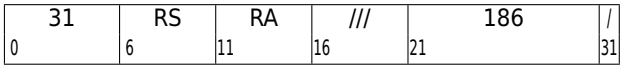
A count of the number of one bits in register RS is placed into register RA. This number ranges from 0 to 64, inclusive.

Special Registers Altered:

None

Parity Doubleword X-form

prtyd RA,RS




```
s ← 0
do i = 0 to 7
   s ← s ⊕ (RS)i×8+7
RA ← 630 || s
```

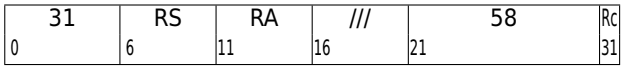
The least significant bit in each byte of the contents of register RS is examined. If there is an odd number of one bits the value 1 is placed into register RA; otherwise the value 0 is placed into register RA.

Special Registers Altered:

None

Count Leading Zeros Doubleword X-form

cntlzd	RA,RS	(Rc=0)
cntlzd.	RA,RS	(Rc=1)




```
n ← 0
do while n < 64
  if (RS)n = 1 then leave
  n ← n + 1
RA ← 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.

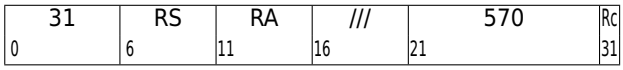
If Rc=1, CR Field 0 is set to reflect the result.

Special Registers Altered:

Register	*Field(s)*	
CR	CR0	(if Rc=1)

Count Trailing Zeros Doubleword X-form

cnttzd	RA,RS	(Rc=0)
cnttzd.	RA,RS	(Rc=1)




```
n  ← 0
do while n < 64
   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 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:

Register	*Field(s)*	
CR	CR0	(if Rc=1)

Count Leading Zeros Doubleword under bit Mask X-form

cntlzdm	RA,RS,RB	


```
count = 0
do i = 0 to 63
   if((RB)i=1) then do
      if((RS)i=1) then break
      count ← count + 1
   end
end
RA ← EXTZ64(count)
```

Let n be the number of bits in register RB having the value 1.

Extract and pack together the contents of the bits in register RS corresponding to a mask specified in register RB, creating an n-bit value.

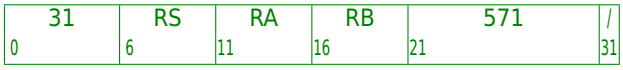
Count the number of contiguous leftmost 0 bits in the n-bit extracted value and place the result into register RA.

Special Registers Altered:

None

Count Trailing Zeros Doubleword under bit Mask X-form

cnttzdm	RA,RS,RB	




```
count ← 0
do i = 0 to 63
   if((RB)63-i=1) then do
      if((RS)63-i=1) then break
      count ← count + 1
   end
end
RA ← EXTZ64(count)
```

Let n be the number of bits in register RB having the value 1.

Extract and pack together the contents of bits in register RS corresponding to a mask specified in register RB, creating an n-bit value.

Count the number of contiguous rightmost 0 bits in the n-bit extracted value and place the result into register RA.

Special Registers Altered:

None

Bit Permute Doubleword X-form

bpermd RA,RS,RB

31	RS	RA	RB	252	/
0	6	11	16	21	31

```
do i = 0 to 7
   index ← (RS)8×i:8×i+7
   If index < 64
      then permi ← (RB)index
      else permi ← 0
RA ← 560 || perm0:7
```

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
```

Centrifuge Doubleword X-form

cfuged RA,RS,RB

31	RS	RA	RB	220	/
0	6	11	16	21	31

```
ptr0 ← 0
ptr1 ← 0
do i = 0 to 63
   if((RB)i=0) then do
      resultptr0 ← (RS)i
      ptr0 ← ptr0 + 1
   end
   if((RB)63-i==1) then do
      result63-ptr1 ← (RS)63-i
      ptr1 ← ptr1 + 1
   end
end
RA ← result
```

The bits in GPR[RS] whose corresponding bits in the mask in GPR[RB] equal 1 are placed in the rightmost bits in GPR[RA] maintaining their relative original order. The other bits in GPR[RS] are placed in the leftmost bits in GPR[RA] maintaining their relative original order.

Special Registers Altered:

None

Parallel Bits Extract Doubleword X-form

pextd RA,RS,RB

31	RS	RA	RB	188	/
0	6	11	16	21	31

```
result ← 0
mask ← (RB)
m ← 0
k ← 0
do while (m < 64)
   if ((RB)63-m == 1) then do
      result63-k ← (RS)63-m
      k ← k + 1
   end
   m ← m + 1
end
RA ← result
```

Let mask be the contents of register RB.

The contents of the bits in register RS corresponding to bits in mask containing a 1 are packed into an n-bit value. The extracted value is placed into register RA.

Special Registers Altered:

None

Parallel Bits Deposit Doubleword X-form

pdepd RA,RS,RB

31	RS	RA	RB	156	/
0	6	11	16	21	31

```
result ← 0
mask ← (RB)
m ← 0
k ← 0
do while (m < 64)
   if (mask63-m == 1) then do
      result63-m ← (RS)63-k
      k ← k + 1
   end
   m ← m + 1
end
RA ← result
```

Let mask be the contents of register RB.

Let n be the number of bits in mask having the value 1.

The contents of the rightmost n bits of register RS are placed into register RA under control of mask as follows.

- The contents of bit 63 of register RS are placed into the bit in register RA corresponding to the rightmost bit in mask that contains a 1,
- the contents of bit 62 of register RS are placed into the bit in register RA corresponding to the second rightmost bit in mask that contains a 1, and so forth until
- the contents of bit 64-n of register RS are placed into the bit in register RA corresponding to the left-most bit in mask that contains a 1.

The contents of bits in register RA corresponding to bits in mask that contain a 0 are set to 0.

Special Registers Altered:

None

3.3.14 Fixed-Point Rotate and Shift Instructions

The Fixed-Point Facility performs rotation operations on data from a GPR and returns the result, or a portion of the result, to a GPR.

The rotation operations rotate a 64-bit quantity left by a specified number of bit positions. Bits that exit from position 0 enter at position 63.

Two types of rotation operation are supported.

For the first type, denoted rotate_{64} or ROTL_{64}, the value rotated is the given 64-bit value. The rotate_{64} operation is used to rotate a given 64-bit quantity.

For the second type, denoted rotate_{32} or ROTL_{32}, 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 rotate_{32} 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:

```
if mstart ≤ mstop then
   mask[mstart:mstop] = ones
   mask[all other bits] = zeros
else
   mask[mstart:63] = ones
   mask[0:mstop] = ones
   mask[all other bits] = zeros
```

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

For instructions that use the rotate_{32} 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 72. *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 954 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.

Architecture Note

For MD-form and MDS-form instructions, the MB and ME fields are used in permuted rather than sequential order because this is easier for the processor. Permuting the MB field permits the processor to obtain the low-order five bits of the MB value from the same place for all instructions having an MB field (M-form and MD-form instructions). Permuting the ME field permits the processor to treat bits 21:26 of all MD-form instructions uniformly.

Rotate Left Word Immediate then AND with Mask M-form

rlwinm	RA,RS,SH,MB,ME	(Rc=0)
rlwinm.	RA,RS,SH,MB,ME	(Rc=1)

21	RS	RA	SH	MB	ME	Rc
0	6	11	16	21	26	31

```
n ← SH
r ← ROTL32((RS)32:63, n)
m ← MASK(MB+32, ME+32)
RA ← r & m
```

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 ANDed with the generated mask and the result is placed into register RA.

Special Registers Altered:

Register	*Field(s)*	
CR	CR0	(if Rc=1)

Extended Mnemonics:

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

Extended mnemonic:		Equivalent to:	
extlwi	RA,RS,n,b	rlwinm	RA,RS,b,0,n−1
srwi	RA,RS,n	rlwinm	RA,RS,32−n,n,31
clrrwi	RA,RS,n	rlwinm	RA,RS,0,0,31−n

Programming Note

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

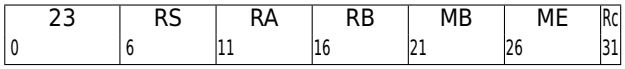
rlwinm can be used to extract an n-bit field that starts at bit position b in RSL, right-justified into the low-order 32 bits of register RA (clearing the remaining 32−n bits of the low-order 32 bits of RA), by setting SH=b+n, MB=32−n, and ME=31. It can be used to extract an n-bit field that starts at 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 = 0, and ME=n−1. It can be used to rotate the contents of the low-order 32 bits of a register left (right) by n bits, by setting SH=n (32−n), MB=0, and ME=31. It can be used to shift the contents of the low-order 32 bits of a register right by n bits, by setting SH=32−n, MB=n, and ME=31. It can be used to clear the high-order b bits of the low-order 32 bits of the contents of a register and then shift the result left by n bits, by setting SH=n, MB=b−n, and ME=31−n. It can be used to clear the low-order n bits of the low-order 32 bits of a register, by setting SH=0, MB=0, and ME=31−n.

For all the uses given above, the high-order 32 bits of register RA are cleared.

Extended mnemonics are provided for all of these uses; see Appendix C, “Assembler Extended Mnemonics” on page 954.

Rotate Left Word then AND with Mask M-form

rlwnm	RA,RS,RB,MB,ME	(Rc=0)
rlwnm.	RA,RS,RB,MB,ME	(Rc=1)




```
n ← (RB)59:63
r ← ROTL32((RS)32:63, n)
m ← MASK(MB+32, ME+32)
RA ← r & m
```

The contents of register RS are rotated$_{32}$ left the number of bits specified by (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:

Register	*Field(s)*	
CR	CR0	(if Rc=1)

Extended Mnemonics:

Extended Mnemonics for *Rotate Left Word then AND with Mask*:

Extended mnemonic:		Equivalent to:	
rotlw	RA,RS,RB	rlwnm	RA,RS,RB,0,31
rotlw.	RA,RS,RB	rlwnm.	RA,RS,RB,0,31

Programming Note

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

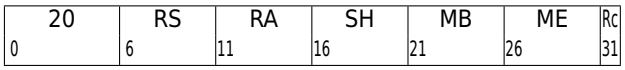
rlwnm can be used to extract an n-bit field that starts at variable bit position b in RSL, right-justified into the low-order 32 bits of register RA (clearing the remaining 32-n bits of the low-order 32 bits of RA), by setting RB$_{59:63}$=b+n, MB=32−n, and ME=31. It can be used to extract an n-bit field 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 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 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 954.

Rotate Left Word Immediate then Mask Insert M-form

rlwimi	RA,RS,SH,MB,ME	(Rc=0)
rlwimi.	RA,RS,SH,MB,ME	(Rc=1)




```
n ← SH
r ← ROTL32((RS)32:63, n)
m ← MASK(MB+32, ME+32)
RA ← r&m | (RA) & ¬m
```

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:

Register	*Field(s)*	
CR	CR0	(if Rc=1)

Extended Mnemonics:

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

Extended mnemonic:		Equivalent to:	
inslwi	RA,RS,n,b	rlwimi	RA,RS,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.

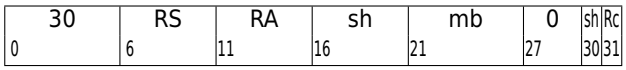
rlwimi can be used to insert an n-bit field that is left-justified in the low-order 32 bits of register RS, into RAL starting at bit position b, by setting SH=32−b, MB=b, and ME=(b+n)−1. It can be used to insert an n-bit field that is right-justified in the low-order 32 bits of register RS, into RAL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

Extended mnemonics are provided for both of these uses; see Appendix C, “Assembler Extended Mnemonics” on page 954.

3.3.14.1.1 64-bit Fixed-Point Rotate Instructions

Rotate Left Doubleword Immediate then Clear Left MD-form

rldicl	RA,RS,SH,MB	(Rc=0)
rldicl.	RA,RS,SH,MB	(Rc=1)




```
SH ← sh5 || sh0:4
r ← ROTL64((RS), SH)
MB ← mb5 || mb0:4
m ← MASK(MB, 63)
RA ← 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:

Register	*Field(s)*	
CR	CR0	(if Rc=1)

Extended Mnemonics:

Extended Mnemonics for *Rotate Left Doubleword Immediate then Clear Left*:

Extended mnemonic:		Equivalent to:	
extrdi	RA,RS,n,b	rldicl	RA,RS,b+n,64-n
srdi	RA,RS,n	rldicl	RA,RS,64-n,n
clrldi	RA,RS,n	rldicl	RA,RS,0,n

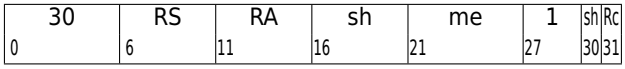
Programming Note

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 954.

Rotate Left Doubleword Immediate then Clear Right MD-form

rldicr	RA,RS,SH,ME	(Rc=0)
rldicr.	RA,RS,SH,ME	(Rc=1)




```
SH ← sh5 || sh0:4
r ← ROTL64((RS), SH)
ME ← me5 || me0:4
m ← MASK(0, ME)
RA ← 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:

Register	*Field(s)*	
CR	CR0	(if Rc=1)

Extended Mnemonics:

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

Extended mnemonic:		Equivalent to:	
extldi	RA,RS,n,b	rldicr	RA,RS,b,n-1
sldi	RA,RS,n	rldicr	RA,RS,n,63-n
clrrdi	RA,RS,n	rldicr	RA,RS,0,63-n

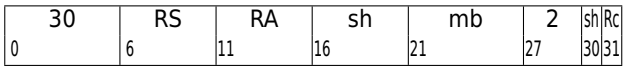
Programming Note

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 954.

Rotate Left Doubleword Immediate then Clear MD-form

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




```
SH ← sh5 || sh0:4
r ← ROTL64((RS), SH)
MB ← mb5 || mb0:4
m ← MASK(MB, 63-SH)
RA ← 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-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:

Register	*Field(s)*	
CR	CR0	(if Rc=1)

Extended Mnemonics:

Extended Mnemonics for *Rotate Left Doubleword Immediate then Clear*:

Extended mnemonic:	Equivalent to:
clrlsldi RA,RS,b,n	rldic RA,RS,n,b-n
clrlsldi.RA,RS,b,n	rldic. RA,RS,n,b-n

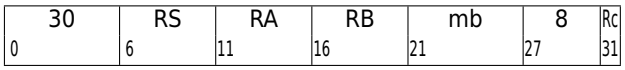
Programming Note

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 ***rldicl***); see Appendix C, “Assembler Extended Mnemonics” on page 954.

Rotate Left Doubleword then Clear Left MDS-form

rldcl	RA,RS,RB,MB	(Rc=0)
rldcl.	RA,RS,RB,MB	(Rc=1)




```
n ← (RB)58:63
r ← ROTL64((RS), n)
MB ← mb5 || mb0:4
m ← MASK(MB, 63)
RA ← r & m
```

The contents of register RS are rotated$_{64}$ 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:

Register	*Field(s)*	
CR	CR0	(if Rc=1)

Extended Mnemonics:

Extended Mnemonics for *Rotate Left Doubleword then Clear Left*:

Extended mnemonic:	Equivalent to:
rotld RA,RS,RB	rldcl RA,RS,RB,0
rotld. RA,RS,RB	rldcl. RA,RS,RB,0

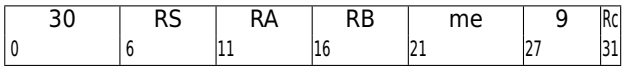
Programming Note

rldcl can be used to extract an n-bit field that starts at variable bit position b in register RS, right-justified into register RA (clearing the remaining 64-n bits of RA), by setting RB$_{58:63}$=b+n and MB=64-n. It can be used to rotate the contents of a register left (right) by variable n bits, by setting RB$_{58:63}$=n (64-n) and MB=0.

Extended mnemonics are provided for some of these uses; see Appendix C, “Assembler Extended Mnemonics” on page 954.

Rotate Left Doubleword then Clear Right MDS-form

rldcr	RA,RS,RB,ME	(Rc=0)
rldcr.	RA,RS,RB,ME	(Rc=1)




```
n ← (RB)58:63
r ← ROTL64((RS), n)
ME ← me5 || me0:4
m ← MASK(0, ME)
RA ← r & m
```

The contents of register RS are rotated$_{64}$ 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:

Register	*Field(s)*	
CR	CR0	(if Rc=1)

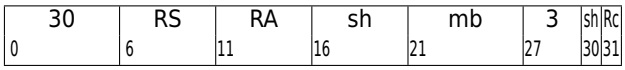
Programming Note

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 954.

Rotate Left Doubleword Immediate then Mask Insert MD-form

rldimi	RA,RS,SH,MB	(Rc=0)
rldimi.	RA,RS,SH,MB	(Rc=1)




```
SH ← sh5 || sh0:4
r ← ROTL64((RS), SH)
MB ← mb5 || mb0:4
m ← MASK(MB, 63-SH)
RA ← r&m | (RA) & ¬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−SH and 0-bits elsewhere. The rotated data are inserted into register RA under control of the generated mask.

Special Registers Altered:

Register	*Field(s)*	
CR	CR0	(if Rc=1)

Extended Mnemonics:

Extended Mnemonics for *Rotate Left Doubleword Immediate then Mask Insert*:

Extended mnemonic:		**Equivalent to:**	
insrdi	RA,RS,b,n	rldimi	RA,RS,64-(b+n),b
insrdi.	RA,RS,b,n	rldimi.	RA,RS,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 954.

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 954 for additional extended mnemonics.

Programming Note

Any Shift Right Algebraic instruction, followed by ***addze***, can be used to divide quickly by 2^n. The setting of the CA and CA32 bits by the Shift Right Algebraic instructions is independent of mode.

Engineering Note

The instructions intended for use with 32-bit data are shown as doing a rotate_{32} operation. This is strictly necessary only for setting the CA and CA32 bits for ***srawi*** and ***sraw***. ***slw*** and ***srw*** could do a rotate_{64} operation if that is easier.

Shift Left Word X-form

slw	RA,RS,RB	(Rc=0)
slw.	RA,RS,RB	(Rc=1)

31	RS	RA	RB	24	Rc
0	6	11	16	21	31

```
n ← (RB)59:63
r ← ROTL32((RS)32:63, n)
if (RB)58 = 0 then
     m ← MASK(32, 63−n)
else m ← 640
RA ← r & m
```

The contents of the low-order 32 bits of register RS are shifted left the number of bits specified by $(RB)_{58:63}$. Bits shifted out of position 32 are lost. Zeros are supplied to the vacated positions on the right. The 32-bit result is placed into $RA_{32:63}$. $RA_{0:31}$ are set to zero. Shift amounts from 32 to 63 give a zero result.

Special Registers Altered:

Register	*Field(s)*	
CR	CR0	(if Rc=1)

Shift Right Word X-form

srw	RA,RS,RB	(Rc=0)
srw.	RA,RS,RB	(Rc=1)

31	RS	RA	RB	536	Rc
0	6	11	16	21	31

```
n ← (RB)59:63
r ← ROTL32((RS)32:63, 64−n)
if (RB)58 = 0 then
    m ← MASK(n+32, 63)
else m ← 640
RA ← r & m
```

The contents of the low-order 32 bits of register RS are shifted right the number of bits specified by $(RB)_{58:63}$. Bits shifted out of position 63 are lost. Zeros are supplied to the vacated positions on the left. The 32-bit result is placed into $RA_{32:63}$. $RA_{0:31}$ are set to zero. Shift amounts from 32 to 63 give a zero result.

Special Registers Altered:

Register	*Field(s)*	
CR	CR0	(if Rc=1)

Shift Right Algebraic Word Immediate X-form

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

31	RS	RA	SH	824	Rc
0	6	11	16	21	31

```
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 $RA_{32:63}$. Bit 32 of RS is replicated to fill $RA_{0:31}$. CA and CA32 are set to 1 if the low-order 32 bits of (RS) contain a negative number and any 1-bits are shifted out of position 63; otherwise CA and CA32 are set to 0. A shift amount of zero causes RA to receive EXTS((RS)32:63), and CA and CA32 to be set to 0.

Special Registers Altered:

Register	*Field(s)*	
XER	CA CA32	
CR	CR0	(if Rc=1)

Shift Right Algebraic Word X-form

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

31	RS	RA	RB	792	Rc
0	6	11	16	21	31

```
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 $RA_{32:63}$. Bit 32 of RS is replicated to fill $RA_{0:31}$. CA and CA32 are set to 1 if the low-order 32 bits of (RS) contain a negative number and any 1-bits are shifted out of position 63; otherwise CA and CA32 are set to 0. A shift amount of zero causes RA to receive EXTS((RS)32:63), and CA and CA32 to be set to 0. Shift amounts from 32 to 63 give a result of 64 sign bits, and cause CA and CA32 to receive the sign bit of $(RS)_{32:63}$.

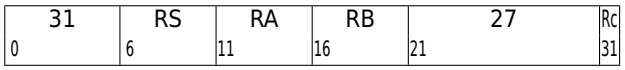
Special Registers Altered:

Register	*Field(s)*	
XER	CA CA32	
CR	CR0	(if Rc=1)

3.3.14.2.1 64-bit Fixed-Point Shift Instructions

Shift Left Doubleword X-form

sld	RA,RS,RB	(Rc=0)
sld.	RA,RS,RB	(Rc=1)




```
n ← (RB)58:63
r ← ROTL64((RS), n)
if (RB)57 = 0 then
     m ← MASK(0, 63-n)
else m ← 64 0
RA ← r & m
```

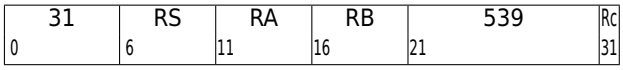
The contents of register RS are shifted left the number of bits specified by (RB)57:63. Bits shifted out of position 0 are lost. Zeros are supplied to the vacated positions on the right. The result is placed into register RA. Shift amounts from 64 to 127 give a zero result.

Special Registers Altered:

Register	*Field(s)*	
CR	CR0	(if Rc=1)

Shift Right Doubleword X-form

srd	RA,RS,RB	(Rc=0)
srd.	RA,RS,RB	(Rc=1)




```
n ← (RB)58:63
r ← ROTL64((RS), 64-n)
if (RB)57 = 0 then
    m ← MASK(n, 63)
else m ← 64 0
RA ← r & m
```

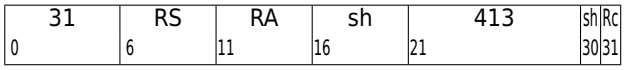
The contents of register RS are shifted right the number of bits specified by (RB)57:63. Bits shifted out of position 63 are lost. Zeros are supplied to the vacated positions on the left. The result is placed into register RA. Shift amounts from 64 to 127 give a zero result.

Special Registers Altered:

Register	*Field(s)*	
CR	CR0	(if Rc=1)

Shift Right Algebraic Doubleword Immediate XS-form

sradi	RA,RS,SH	(Rc=0)
sradi.	RA,RS,SH	(Rc=1)




```
SH ← sh5 || sh0:4
r ← ROTL64((RS), 64-SH)
m ← MASK(SH, 63)
s ← (RS)0
RA ← r&m | (64s) & ¬m
carry ← s & ((r & ¬m) ≠ 0)
CA    ← carry
CA32  ← carry
```

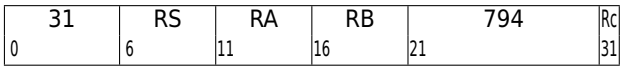
The contents of register RS are shifted right SH bits. Bits shifted out of position 63 are lost. Bit 0 of RS is replicated to fill the vacated positions on the left. The result is placed into register RA. CA and CA32 are set to 1 if (RS) is negative and any 1-bits are shifted out of position 63; otherwise 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:

Register	*Field(s)*	
XER	CA CA32	
CR	CR0	(if Rc=1)

Shift Right Algebraic Doubleword X-form

srad	RA,RS,RB	(Rc=0)
srad.	RA,RS,RB	(Rc=1)




```
n ← (RB)58:63
r ← ROTL64((RS), 64-n)
if (RB)57 = 0 then
    m ← MASK(n, 63)
else m ← 64 0
s ← (RS)0
RA ← r&m | (64s) & ¬m
carry ← s & ((r & ¬m) ≠ 0)
CA    ← carry
CA32  ← carry
```

The contents of register RS are shifted right the number of bits specified by (RB)57:63. Bits shifted out of position 63 are lost. Bit 0 of RS is replicated to fill the vacated positions on the left. The result is placed into register RA. CA and CA32 are set to 1 if (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. Shift amounts from 64 to 127 give a result of 64 sign bits in RA, and cause CA and CA32 to receive the sign bit of (RS).

Special Registers Altered:

Register	*Field(s)*	
XER	CA CA32	
CR	CR0	(if Rc=1)

Extend Sign Word and Shift Left Immediate XS-form

extswsli	RA,RS,SH	(Rc=0)
extswsli.	RA,RS,SH	(Rc=1)

31	RS	RA	sh	445	sh	Rc
0	6	11	16	21	30	31

```
SH ← sh5 || sh0:4
r  ← ROTL64(EXTS64(RS32:63), SH)
m  ← MASK(0, 63-SH)
RA ← r & 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:

Register	*Field(s)*	
CR	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 (***cdtbcd***) See 5 for additional information.

Convert Declets To Binary Coded Decimal X-form

cdtbcd RA,RS

31	RS	RA	///	282	/
0	6	11	16	21	31

```
do i = 0 to 1
   n ← i × 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

31	RS	RA	///	314	/
0	6	11	16	21	31

```
do i = 0 to 1
   n ← i × 32
      RA_{n+0:n+11}  ← 0
      RA_{n+12:n+21} ← BCD_TO_DPD((RS)_{n+8:n+19})
      RA_{n+22:n+31} ← BCD_TO_DPD((RS)_{n+20:n+31})
```

The low-order 24 bits of each word of register RS contain six, 4-bit BCD fields which are converted to two declets; each set of two declets is placed into the low-order 20 bits of the corresponding word in RA. The high-order 12 bits in each word of RA are set to 0.

If a 4-bit BCD field has a value greater than 9 the results are undefined.

Special Registers Altered:

None

Add and Generate Sixes XO-form

addg6s RT,RA,RB

31	RT	RA	RB	/	74	/
0	6	11	16	21	22	31

```
do i = 0 to 15
   dc_i ← carry_out(RA_{4×i:63} + RB_{4×i:63})
c ← 4(dc_0) || 4(dc_1) || ... || 4(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 4×n).

A doubleword is composed from the 16 carry bits, and placed into RT. The doubleword consists of a decimal six (0b0110) in every decimal digit position for which the corresponding carry bit is 0, and a zero (0b0000) in every position for which the corresponding carry bit is 1.

Special Registers Altered:

None

Programming Note

addg6s can be used to add or subtract two BCD operands. In these examples it is assumed that r0 contains 0x666...666. (BCD data formats are described in Section 5.3.)

Addition of the unsigned BCD operand in register RA to the unsigned BCD operand in register RB can be accomplished as follows.

```
add     r1,RA,r0
add     r2,r1,RB
addg6s  RT,r1,RB
subf    RT,RT,r2   # RT = RA +_BCD RB
```

Subtraction of the unsigned BCD operand in register RA from the unsigned BCD operand in register RB can be accomplished as follows. (In this example it is assumed that RB is not register 0.)

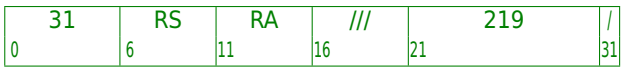
```
addi    r1,RB,1
nor     r2,RA,RA   # one's complement of RA
add     r3,r1,r2
addg6s  RT,r1,r2
subf    RT,RT,r3   # RT = RB −_BCD RA
```

Additional instructions are needed to handle signed BCD operands, and BCD operands that occupy more than one register (e.g., unsigned BCD operands that have more than 16 decimal digits).

3.3.16 Byte-Reverse Instructions

Byte-Reverse Halfword X-form

brh RA,RS




```
RA ← (RS)8:15  || (RS)0:7   ||
     (RS)24:31 || (RS)16:23 ||
     (RS)40:47 || (RS)32:39 ||
     (RS)56:63 || (RS)48:55
```

The contents of bits 0:15 of register RS are placed into bits 0:15 of register RA in byte-reversed order.

The contents of bits 16:31 of register RS are placed into bits 16:31 of register RA in byte-reversed order.

The contents of bits 32:47 of register RS are placed into bits 32:47 of register RA in byte-reversed order.

The contents of bits 48:63 of register RS are placed into bits 48:63 of register RA in byte-reversed order.

Special Registers Altered:

None

Byte-Reverse Word X-form

brw RA,RS


```
RA ← (RS)24:31 || (RS)16:23 || (RS)8:15  || (RS)0:7   ||
     (RS)56:63 || (RS)48:55 || (RS)40:47 || (RS)32:39
```

The contents of bits 0:31 of register RS are placed into bits 0:31 of register RA in byte-reversed order.

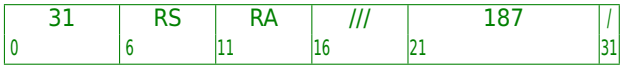
The contents of bits 32:63 of register RS are placed into bits 32:63 of register RA in byte-reversed order.

Special Registers Altered:

None

Byte-Reverse Doubleword X-form

brd RA,RS




```
RA ← (RS)56:63 || (RS)48:55 || (RS)40:47 || (RS)32:39 ||
     (RS)24:31 || (RS)16:23 || (RS)8:15  || (RS)0:7
```

The contents of register RS are placed into register RA in byte-reversed order.

Special Registers Altered:

None

3.3.17 Fixed-Point Hash Instructions

The *Hash* instructions compute a doubleword hash value using a predefined hash function on two doublewords from two GPRs and a doubleword key provided by an SPR having higher privilege than the privilege of the instruction. The *Hash Store* [*Privileged*] instructions store the computed doubleword hash value to a doubleword storage location defined by one of the input GPRs and an immediate field of the instruction. The *Hash Check* [*Privileged*] instructions read the doubleword from the storage location specified in the same fashion with a GPR and an immediate field, compare the computed hash value with the value read, and invoke the system trap handler if the values do not match. The hash function used for these instructions is described in the following subsection. The privileged forms of these instructions are described in Section 5.4.2 of Book III.

Programming Note

The opcodes used by the *Hash* instructions were used for reserved-no-ops in versions of the architecture that precede Version 3.1B (and are subsequent to V. 2.04). Therefore programs that will run on processors that comply with any of these preceding versions can use the *Hash* instructions, and will obtain the benefits of the hash store and check functions when the programs run on processors that comply with V. 3.1B or a subsequent version.

3.3.17.1 Hash Function Description

SIMON_LIKE_32_64(x, key,lane)

Let x be a word and key be a doubleword. lane is an unsigned integer. result is a word. c is a halfword constant. Z0 is a doubleword constant. Z and temp are halfwords. k[i], eff_k[i], xleft[i], xright[i], fxleft[i] are all halfwords for all values of i = 0 through 32.

```
c ← 0xFFFC
Z0 ← 0xFA25_61CD_F44A_C398
     // = 0x3E89_5873_7D12_B0E6<<2
Z ← 0x0000
k[0] ← key.hword[0]
k[1] ← key.hword[1]
k[2] ← key.hword[2]
k[3] ← key.hword[3]
xleft[0] ← x.hword[1]
xright[0] ← x.hword[0]
for i = 0 to 27
  Z.bit[15] ← Z0.bit[i]
  temp ←(k[i + 3] >>> 3) ⊕ k[i + 1]
  k[i + 4] ← c ⊕ Z ⊕ k[i] ⊕ temp ⊕ (temp >>> 1)
end
for i = 0 to 7
  eff_k[4*i]   ← k[4*i + ((0 + lane) mod 4)]
  eff_k[4*i+1] ← k[4*i + ((1 + lane) mod 4)]
  eff_k[4*i+2] ← k[4*i + ((2 + lane) mod 4)]
  eff_k[4*i+3] ← k[4*i + ((3 + lane) mod 4)]
end
for i = 0 to 31
 fxleft[i] ← ((xleft[i]<<<1) & (xleft[i]<<<8)) ⊕
             (xleft[i]<<<2)
 xleft[i + 1] ← xright[i] ⊕ fxleft[i] ⊕ eff_k[i]
 xright[i + 1] ← xleft[i]
end
result.hword[1] ← xleft[32]
result.hword[0] ← xright[32]
return(result)
```

HashDigest(x, y, key)

Let x,y, and key be doublewords. stage0 and stage1 are quadwords. stage2left, stage2right, and result are doublewords.

```
for i = 0 to 7
  stage0.byte[2*i] ←  y.byte[7 - i]
  stage0.byte[2*i + 1] ← x.byte[i]
end
for i = 0 to 3
  stage1.word[i] ← SIMON_LIKE_32_64(stage0.word[i],
                                    key, i)
end
stage2left.word[0] ← stage1.word[0]
stage2left.word[1] ← stage1.word[1]
stage2right.word[0] ← stage1.word[2]
stage2right.word[1] ← stage1.word[3]
result ← stage2left ⊕ stage2right

return(result)
```

Programming Note

The hash function is similar to the SIMON cipher hash function with block size 32 bits and key size 64 bits described in the academic paper “The SIMON and SPECK lightweight block ciphers” In Proceedings of the 52nd Annual Design Automation Conference authored by Ray Beaulieu, Douglas Shors, Jason Smith, Stefan Treatman-Clark, Bryan Weeks, and Louis Wingers, with some differences regarding key schedule and digest computation.

3.3.17.2 Fixed-Point Hash Instructions

Hash Store X-form

hashst RB,offset(RA)

31	D	RA	RB	722	DX
0	6	11	16	21	31

```
DW ← 32×DX + D
d ← EXTS(0b111_1111 || DW || 0b000)
EA ← (RA) + d
temp ← HashDigest((RA), (RB), HASHKEYR)
MEM(EA, 8) ← temp
```

Let DW be the value 32×DX + D. The offset is (0b111_1111 || DW || 0b000) sign extended to 64 bits. Let the effective address (EA) be the sum (RA) + offset. The doubleword hash value computed from the contents of RA, RB, and the privileged SPR HASHKEYR, as specified by the HashDigest function described in Section 3.3.17.1 of Book I, is stored into the doubleword in storage addressed by EA.

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.

If RA=0, the instruction form is invalid.

Special Registers Altered:

None

Hash Check X-form

hashchk RB,offset(RA)

31	D	RA	RB	754	DX
0	6	11	16	21	31

```
DW ← 32×DX + D
d ← EXTS(0b111_1111 || DW || 0b000)
EA ← (RA) + d
temp ← HashDigest((RA), (RB), HASHKEYR)
temp1 ← MEM(EA, 8)
if (temp ≠ temp1) then TRAP
```

Let DW be the value 32×DX + D. The offset is (0b111_1111 || DW || 0b000) sign extended to 64 bits. Let the effective address (EA) be the sum (RA) + offset. The doubleword in storage addressed by EA is read and compared with the doubleword hash value computed from the contents of RA, RB, and the privileged SPR HASHKEYR, as specified by the HashDigest function described in Section 3.3.17.1 of Book I. If the values are unequal, the system trap handler is invoked.

This instruction is treated as a *Load*; see Section 4.3 of Book II.

If the values are unequal, this instruction is context synchronizing (see Book III).

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.

If RA=0, the instruction form is invalid.

Special Registers Altered:

None

Programming Note

If RA=0 the instruction form is invalid because otherwise, for architecture consistency, the EA computation would use (RA|0) instead of (RA), and if RA=0 neither alternative – (RA) or 0 – would make sense as the first argument of the HashDigest function.

- If the contents of GPR 0 were used as the first argument, the contents of GPR 0 would have to remain constant between the ***hashst*** and the ***hashchk*** (or be saved and restored between the two instructions), which is not normally the case when GPR 0 is used as RA in EA specification.
- If 0 were used as the first argument, security would be reduced because the first argument would be a constant.

(In normal use of the two instructions the hash value will be stored in the stack frame, and GPR 0 is never used as the stack frame pointer, so nothing is lost in practice by preventing GPR 0 from being used as RA.)

Programming Note

A primary use of the *Hash* instructions is to enable programs to protect against corruption of the saved value of the Link Register in the program stack and thereby to provide Return Oriented Programming (ROP) protection. The ***hashst*** and ***hashchk*** instructions can serve this purpose for application programs, while the ***hashstp*** and ***hashchkp*** instructions can do so for privileged programs (see Section 5.4.2 of Book III).

On being called the callee would store a hash of the Link Register contents and the stack pointer in the following fashion:

```
mflr    r0              # move LR to GPR 0
std     r0,offset1(r1)  # save LR in stack
hashst  r0,offset2(r1)  # save hash in stack
```

Before return the callee would restore the Link Register and recompute the hash of the Link Register contents and the stack pointer and compare the result with the hash value previously stored in the stack in the following fashion:

```
ld      r0,offset1(r1) # restore LR value
hashchk r0,offset2(r1) # recompute hash, check vs memory contents at (r1) + offset2
                       # and trap if they mismatch
mtlr    r0             # restore LR
blr                    # return
```

Note that the EA specified by the *Hash* instructions — (RA) + EXTS(0b111_1111 || DW || 0b000) — can be expressed as (RA) − (64−DW)×8. Therefore the Hash instructions can only access one of the 64 doublewords preceding the address provided by RA (from (RA) − 8 to (RA) − 512). Assuming RA is the stack pointer, as in the program fragments above, the stack needs to have spare storage in that range for this usage pattern.

Also note that in the above program pattern if the *Hash* instructions are treated as no-ops during program execution then the program still executes correctly albeit without the additional check on corruption of the Link Register value stored in the stack. This property permits dynamic runtime disablement of this check by setting the NPHIE (Non-Privileged Hash Instruction Enable) and PHIE (Privileged Hash Instruction Enable) bits to 0 in the DEXCR and HDEXCR. (See Section 9.3 of Book III.)

Programming Note

As part of creating a process, operating systems should generate a new key value using either a hardware random number generator (e.g. the ***darn*** instruction) or a software pseudo-random number generator and assign it to the HASHKEYR SPR. Thereafter the value should be maintained as part of the process context.

The hypervisor should have analogous behavior with regards to the HASHPKEYR SPR and partition context.

The two SPRs are described in Section 5.3.8 of Book III.

3.3.18 Move To/From Vector-Scalar Register Instructions

Move From VSR Doubleword X-form

mfvsrd RA,XS

31	S	RA	///	51	SX
0	6	11	16	21	31

```
if SX=0 & MSR.FP=0 then FP_Unavailable()
if SX=1 & MSR.VEC=0 then Vector_Unavailable()

GPR[RA] ← VSR[32×SX+S].dword[0]
```

Let XS be the value 32×SX + S.

The contents of doubleword element 0 of VSR[XS] are placed into GPR[RA].

For SX=0, ***mfvsrd*** is treated as a *Floating-Point* instruction in terms of resource availability.

For SX=1, ***mfvsrd*** is treated as a *Vector* instruction in terms of resource availability.

Special Registers Altered:

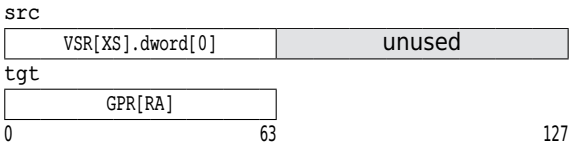
None

Extended Mnemonics:

Extended Mnemonics for *Move From VSR Doubleword*:

Extended mnemonic:		Equivalent to:	
mffprd	RA,frs	mfvsrd	RA,frs
mfvrd	RA,vrs	mfvsrd	RA,vrs+32

Data Layout for mfvsrd



Move From VSR Lower Doubleword X-form

mfvsrld RA,XS

31	S	RA	///	307	SX
0	6	11	16	21	31

```
if SX=0 & MSR.VSX=0 then VSX_Unavailable()
if SX=1 & MSR.VEC=0 then Vector_Unavailable()

GPR[RA] ← VSR[32×SX+S].dword[1]
```

Let XS be the value 32×SX + S.

The contents of doubleword 1 of VSR[XS] are placed into GPR[RA].

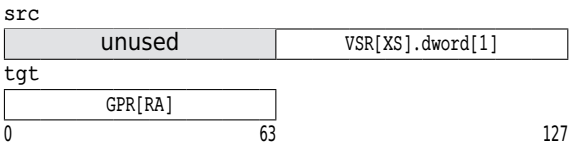
For SX=0, ***mfvsrld*** is treated as a *VSX* instruction in terms of resource availability.

For SX=1, ***mfvsrld*** is treated as a *Vector* instruction in terms of resource availability.

Special Registers Altered:

None

Data Layout for mfvsrld



Move From VSR Word and Zero X-form

mfvsrwz RA,XS

31	S	RA	///	115	SX
0	6	11	16	21	31

```
if SX=0 & MSR.FP=0 then FP_Unavailable()
if SX=1 & MSR.VEC=0 then Vector_Unavailable()

GPR[RA] ← EXTZ64(VSR[32×SX+S].word[1])
```

Let XS be the value 32×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.

Special Registers Altered:

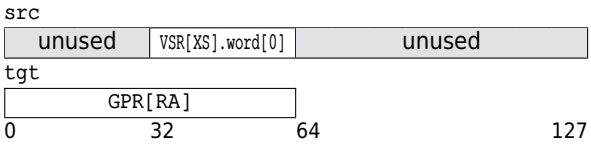
None

Extended Mnemonics:

Extended Mnemonics for *Move To VSR Word and Zero*:

Extended mnemonic:		Equivalent to:	
mffprwz	RA,frs	mfvsrwz	RA,frs
mfvrwz	RA,vrs	mfvsrwz	RA,vrs+32

Data Layout for mfvsrwz



Move To VSR Doubleword X-form

mtvsrd XT,RA

31	T	RA	///	179	TX
0	6	11	16	21	31

```
if TX=0 & MSR.FP=0 then FP_Unavailable()
if TX=1 & MSR.VEC=0 then Vector_Unavailable()

VSR[32×TX+T].dword[0] ← GPR[RA]
VSR[32×TX+T].dword[1] ← 0xUUUU_UUUU_UUUU_UUUU
```

Let XT be the value 32×TX + T.

The contents of GPR[RA] are placed into doubleword element 0 of VSR[XT].

The contents of doubleword element 1 of VSR[XT] are undefined.

For TX=0, ***mtvsrd*** is treated as a *Floating-Point* instruction in terms of resource availability.

For TX=1, ***mtvsrd*** is treated as a *Vector* instruction in terms of resource availability.

Special Registers Altered:

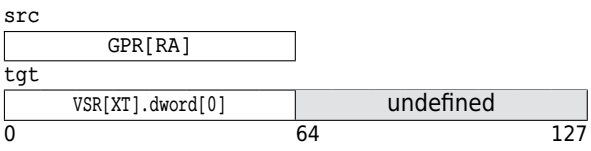
None

Extended Mnemonics:

Extended Mnemonics for *Move To VSR Doubleword*:

Extended mnemonic:		Equivalent to:	
mtfprd	frt,RA	mtvsrd	frt,RA
mtvrd	vrt,RA	mtvsrd	vrt+32,RA

Data Layout for mtvsrd



Move To VSR Word Algebraic X-form

mtvsrwa XT,RA

31	T	RA	///	211	TX
0	6	11	16	21	31

```
if TX=0 & MSR.FP=0 then FP_Unavailable()
if TX=1 & MSR.VEC=0 then Vector_Unavailable()

VSR[32×TX+T].dword[0] ← EXTS64(GPR[RA].bit[32:63])
VSR[32×TX+T].dword[1] ← 0xUUUU_UUUU_UUUU_UUUU
```

Let XT be the value 32×TX + T.

The two's-complement integer in bits 32:63 of GPR[RA] is sign-extended to 64 bits and placed into doubleword element 0 of VSR[XT].

The contents of doubleword element 1 of VSR[XT] are undefined.

For TX=0, ***mtvsrwa*** is treated as a *Floating-Point* instruction in terms of resource availability.

For TX=1, ***mtvsrwa*** is treated as a *Vector* instruction in terms of resource availability.

Special Registers Altered:

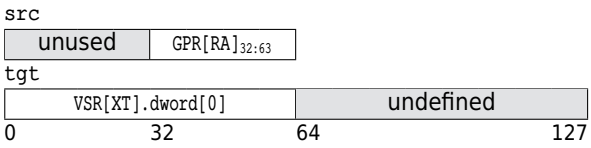
None

Extended Mnemonics:

Extended Mnemonics for *Move To VSR Word Algebraic*:

Extended mnemonic:		Equivalent to:	
mtfprwa	frt,RA	mtvsrwa	frt,RA
mtvrwa	vrt,RA	mtvsrwa	vrt+32,RA

Data Layout for mtvsrwa



Move To VSR Word and Zero X-form

mtvsrwz XT,RA

31	T	RA	///	243	TX
0	6	11	16	21	31

```
if TX=0 & MSR.FP=0 then FP_Unavailable()
if TX=1 & MSR.VEC=0 then Vector_Unavailable()

VSR[32×TX+T].dword[0] ← EXTZ64(GPR[RA].word[1])
VSR[32×TX+T].dword[1] ← 0xUUUU_UUUU_UUUU_UUUU
```

Let XT be the value 32×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.

Special Registers Altered:

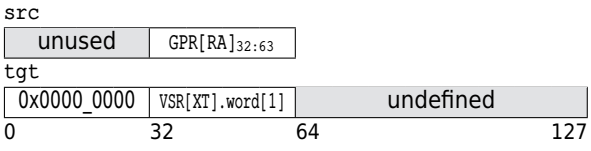
None

Extended Mnemonics:

Extended Mnemonics for *Move To VSR Word and Zero*:

Extended mnemonic:		Equivalent to:	
mtfprwz	frt,RA	mtvsrwz	frt,RA
mtvrwz	vrt,RA	mtvsrwz	vrt+32,RA

Data Layout for mtvsrwz



Move To VSR Double Doubleword X-form

mtvsrdd XT,RA,RB

31	T	RA	RB	435	TX
0	6	11	16	21	31

```
if TX=0 & MSR.VSX=0 then VSX_Unavailable()
if TX=1 & MSR.VEC=0 then Vector_Unavailable()

if RA=0 then
   VSR[32×TX+T].dword[0] ← 0x0000_0000_0000_0000
else
   VSR[32×TX+T].dword[0] ← GPR[RA]

VSR[32×TX+T].dword[1] ← GPR[RB]
```

Let XT be the value 32×TX + T.

The contents of GPR[RA], or the value 0 if RA=0, are placed into doubleword 0 of VSR[XT].

The contents of GPR[RB] are placed into doubleword 1 of VSR[XT].

For TX=0, ***mtvsrdd*** is treated as a *VSX* instruction in terms of resource availability.

For TX=1, ***mtvsrdd*** is treated as a *Vector* instruction in terms of resource availability.

Special Registers Altered:

None

Data Layout for mtvsrdd

src.dword[0]

GPR[RA]

src.dword[1]

GPR[RB]

tgt

VSR[XT].dword[0]	VSR[XT].dword[1]
0	64	127

Move To VSR Word & Splat X-form

mtvsrws XT,RA

31	T	RA	///	403	TX
0	6	11	16	21	31

```
if TX=0 & MSR.VSX=0 then VSX_Unavailable()
if TX=1 & MSR.VEC=0 then Vector_Unavailable()

VSR[32×TX+T].word[0] ← GPR[RA].bit[32:63]
VSR[32×TX+T].word[1] ← GPR[RA].bit[32:63]
VSR[32×TX+T].word[2] ← GPR[RA].bit[32:63]
VSR[32×TX+T].word[3] ← GPR[RA].bit[32:63]
```

Let XT be the value 32×TX + T.

The contents of bits 32:63 of GPR[RA] are placed into each word element of VSR[XT].

For TX=0, ***mtvsrws*** is treated as a *VSX* instruction in terms of resource availability.

For TX=1, ***mtvsrws*** is treated as a *Vector* instruction in terms of resource availability.

Special Registers Altered:

None

Data Layout for mtvsrws

src

unused	$GPR[RA]_{32:63}$

tgt

VSR[XT].word[0]	VSR[XT].word[1]	VSR[XT].word[2]	VSR[XT].word[3]
0	32	64	96	127

3.3.19 Move To/From System Register Instructions

The *Move To Condition Register Fields* instruction has a preferred form; see Section 1.8.1, “Preferred Instruction Forms” on page 25. In the preferred form, the FXM field satisfies the following rule.

- Exactly one bit of the FXM field is set to 1.

Extended mnemonics

Extended mnemonics are provided for the ***mtspr*** and ***mfspr*** instructions so that they can be coded with the SPR name as part of the mnemonic rather than as a numeric operand. An extended mnemonic is provided for the ***mtcrf*** instruction for compatibility with old software (written for a version of the architecture that 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 954 for additional extended mnemonics.

Move To Special Purpose Register XFX-form

mtspr SPR,RS

31	RS	spr	467	/
0	6	11	21	31

```
n ← spr5:9 || spr0:4
switch(n)
   case(13): see Book III
   case(800, 801, 802, 803): see Book II
   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 Table 3.1 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), or 800–803 (denoting aliases for the BESCR or BESCRU), the contents of register RS, or of the low-order 32 bits of register RS if the designated Special Purpose Register is 32 bits long, are placed into the designated Special Purpose Register.

The AMR (Authority Mask Register) is used for “storage protection.” This use, and operation of ***mtspr*** for the AMR, are described in Book III.

BESCRS[U] and BESCRR[U] are aliases for the BESCR (Branch Event Status and Control Register) or for the high-order 32 bits thereof. The BESCR is used in conjunction with “event-based branches”, as described in Book II Section 6.2.1. The operation of mtspr for BESCRS[U] and BESCRR[U] is described in Book III.

If execution of this instruction is attempted specifying an SPR number that is not shown in Table 3.1, one of the following occurs.

- If $spr_0 = 0$, the illegal instruction error handler is invoked.
- If $spr_0 = 1$, the system privileged instruction error handler is invoked.

A complete description of this instruction can be found in Book III.

Special Registers Altered:

See above.

Extended Mnemonics:

Examples of extended mnemonics for *Move To Special Purpose Register*:

Extended mnemonic:		Equivalent to:	
mtxer	RS	mtspr	1,RS
mtlr	RS	mtspr	8,RS
mtctr	RS	mtspr	9,RS
mtppr	RS	mtspr	896,RS
mtppr32	RS	mtspr	898,RS

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. This maintains compatibility with POWER SPR encodings, in which these two instructions have only a 5-bit SPR field occupying bits 11:15.

Engineering Note

See the description of the ***mtspr*** and ***mfspr*** instructions in Book III.

decimal	SPR[1] $spr_{5:9}$ $spr_{0:4}$	Register Name
1	00000 00001	XER
3	00000 00011	DSCR
8	00000 01000	LR
9	00000 01001	CTR
13	00000 01101	AMR
256	01000 00000	VRSAVE
769	11000 00001	MMCR2
770	11000 00010	MMCRA
771	11000 00011	PMC1
772	11000 00100	PMC2
773	11000 00101	PMC3
774	11000 00110	PMC4
775	11000 00111	PMC5
776	11000 01000	PMC6
779	11000 01011	MMCR0
800	11001 00000	BESCRS
801	11001 00001	BESCRSU
802	11001 00010	BESCRR
803	11001 00011	BESCRRU
804	11001 00100	EBBHR
805	11001 00101	EBBRR
806	11001 00110	BESCR
808	11001 01000	reserved[2]
809	11001 01001	reserved[2]
810	11001 01010	reserved[2]
811	11001 01011	reserved[2]
815	11001 01111	TAR
896	11100 00000	PPR
898	11100 00010	PPR32

1. Note that the order of the two 5-bit halves of the SPR number is reversed.
2. Accesses to these registers are no-ops; see Section 1.3.3, “Reserved Fields, Reserved Values, and Reserved SPRs”

Table 3.1: SPR Encodings

Move From Special Purpose Register XFX-form

mfspr RT,SPR

31	RT	spr	339	/
0	6	11	21	31

```
n ← spr5:9 || spr0:4
switch(n)
   case(800, 802): RT ← BESCR
   case(801, 803): RT ← 320 || BESCRU
   case(808, 809, 810, 811):
   default:
      if length(SPR(n)) = 64 then
         RT ← SPR(n)
      else
         RT ← 320 || SPR(n)
```

The SPR field denotes a Special Purpose Register, encoded as shown in the table below. If the SPR field contains a value from 808 through 811, the instruction specifies a reserved SPR, and is treated as a no-op; see Section 1.3.3, “Reserved Fields, Reserved Values, and Reserved SPRs”. Otherwise, the contents of the designated Special Purpose Register are placed into register RT. For Special Purpose Registers that are 32 bits long, the low-order 32 bits of RT receive the contents of the Special Purpose Register and the high-order 32 bits of RT are set to zero.

If the SPR field contains the value 800 (BESCRS) or 802 (BESCRR), the designated SPR is the BESCR. If the SPR field contains the value 801 (BESCRSU) or 803 (BESCRRU), the designated SPR is the BESCRU.

If execution of this instruction is attempted specifying an SPR number that is not shown above, one of the following occurs.

- If $spr_0 = 0$, the illegal instruction error handler is invoked.
- If $spr_0 = 1$, 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:

Extended Mnemonics for *Move From Special Purpose Register*:

Extended mnemonic:		Equivalent to:	
mfxer	RT	mfspr	RT,1
mflr	RT	mfspr	RT,8
mfctr	RT	mfspr	RT,9

decimal	SPR[1] $spr_{5:9}$ $spr_{0:4}$	Register Name
1	00000 00001	XER
3	00000 00011	DSCR
8	00000 01000	LR
9	00000 01001	CTR
13	00000 01101	AMR
136	00100 01000	CTRL
256	01000 00000	VRSAVE
259	01000 00011	SPRG3
268	01000 01100	TB[2]
269	01000 01101	TBU[2]
455	01110 00111	HDEXCR
736	10111 00000	SIER2
737	10111 00001	SIER3
738	10111 00010	MMCR3
768	11000 00000	SIER
769	11000 00001	MMCR2
770	11000 00010	MMCRA
771	11000 00011	PMC1
772	11000 00100	PMC2
773	11000 00101	PMC3
774	11000 00110	PMC4
775	11000 00111	PMC5
776	11000 01000	PMC6
779	11000 01011	MMCR0
780	11000 01100	SIAR
781	11000 01101	SDAR
782	11000 01110	MMCR1
800	11001 00000	BESCRS
801	11001 00001	BESCRSU
802	11001 00010	BESCRR
803	11001 00011	BESCRRU
804	11001 00100	EBBHR
805	11001 00101	EBBRR
806	11001 00110	BESCR
808	11001 01000	reserved[3]
809	11001 01001	reserved[3]
810	11001 01010	reserved[3]
811	11001 01011	reserved[3]
812	11001 01100	DEXCR
815	11001 01111	TAR
896	11100 00000	PPR
898	11100 00010	PPR32

1. Note that the order of the two 5-bit halves of the SPR number is reversed.
2. See Chapter 5 of Book II
3. Accesses to these SPRs are no-ops; see Section 1.3.3, “Reserved Fields, Reserved Values, and Reserved SPRs”.

Note

See the Notes that appear with ***mtspr***.

Move to CR from XER Extended X-form

mcrxrx BF

31	BF	//	///	///	576	/
0	6	9	11	16	21	31

$CR_{4\times BF+32:4\times BF+35} \leftarrow XER_{OV\ OV32\ CA\ CA32}$

The contents of the OV, OV32, CA, and CA32 are copied to Condition Register field BF.

Special Registers Altered:

Register	*Field(s)*
CR	field BF

Move To One Condition Register Field XFX-form

mtocrf FXM,RS

31	RS	1	FXM	/	144	/
0	6	11	12	20	21	31

```
count ← 0
do i = 0 to 7
   if FXM_i = 1 then
      n ← i
   count ← count + 1
if count = 1 then
   CR_{4×n+32:4×n+35} ← (RS)_{4×n+32:4×n+35}
else
   CR ← undefined
```

If exactly one bit of the FXM field is set to 1, let n be the position of that bit in the field ($0 \leq n \leq 7$). 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:

Register	*Field(s)*
CR	field selected by FXM

Move To Condition Register Fields XFX-form

mtcrf FXM,RS

31	RS	0	FXM	/	144	/
0	6	11	12	20	21	31

```
mask ← 4(FXM_0) || 4(FXM_1) || ... 4(FXM_7)
CR ← ((RS)_{32:63} & mask) | (CR & ¬mask)
```

The contents of bits 32:63 of register RS are placed into the Condition Register under control of the field mask specified by FXM. The field mask identifies the 4-bit fields affected. Let i be an integer in the range 0-7. If 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:

Register	*Field(s)*
CR	fields selected by FXM

Extended Mnemonics:

Extended Mnemonics for *Move To Condition Register Fields*:

Extended mnemonic:		**Equivalent to:**	
mtcr	RS	mtcrf	0xFF,RS

Move From One Condition Register Field XFX-form

mfocrf RT,FXM


```
RT ← undefined
count ← 0
do i = 0 to 7
   if FXM_i = 1 then
      n ← i
      count ← count + 1
if count = 1 then
   RT ← 64 0
   RT_4×n+32:4×n+35 ← CR_4×n+32:4×n+35
```

If exactly one bit of the FXM field is set to 1, let n be the position of that bit in the field ($0 \leq n \leq 7$). The contents of CR field n (CR bits 4×n+32:4×n+35) are placed into bits 4×n+32:4×n+35 of register RT, and the remaining bits of register RT are set to zero. Otherwise, the contents of register RT are undefined.

[]

Special Registers Altered:

None

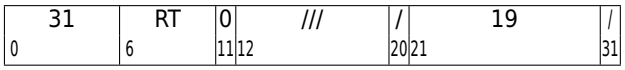
Programming Note

Warning: ***mfocrf*** is not backward compatible with processors that comply with versions of the architecture that precede Version 3.0. 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

mfcr RT




```
RT ← 32 0 || CR
```

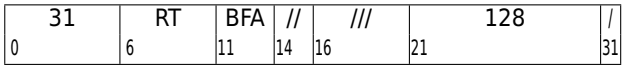
The contents of the Condition Register are placed into $RT_{32:63}$. $RT_{0:31}$ are set to 0.

Special Registers Altered:

None

Set Boolean X-form

setb RT,BFA




```
if CR4×BFA+32=1 then
   RT ← 0xFFFF_FFFF_FFFF_FFFF

else if CR4×BFA+33=1 then
   RT ← 0x0000_0000_0000_0001

else
   RT ← 0x0000_0000_0000_0000
```

If the contents of bit 0 of CR field BFA are equal to 0b1, the contents of register RT are set to 0xFFFF_FFFF_FFFF_FFFF.

Otherwise, if the contents of bit 1 of CR field BFA are equal to 0b1, the contents of register RT are set to 0x0000_0000_0000_0001.

Otherwise, the contents of register RT are set to 0x0000_0000_0000_0000.

Special Registers Altered:

None

Set Boolean Condition X-form

setbc RT,BI


```
RT = (CRBI=1) ?  1 :  0
```

If bit BI of the CR contains a 1, register RT is set to 1. Otherwise, register RT is set to 0.

Special Registers Altered:

None

Set Boolean Condition Reverse X-form

setbcr RT,BI


```
RT = (CRBI=1) ?  0 :  1
```

If bit BI of the CR contains a 1, register RT is set to 0. Otherwise, register RT is set to 1.

Special Registers Altered:

None

Set Negative Boolean Condition X-form

setnbc RT,BI


```
RT = (CRBI=1) ? -1 :  0
```

If bit BI of the CR contains a 1, register RT is set to -1. Otherwise, register RT is set to 0.

Special Registers Altered:

None

Set Negative Boolean Condition Reverse X-form

setnbcr RT,BI


```
RT = (CRBI=1) ?  0 : -1
```

If bit BI of the CR contains a 1, register RT is set to 0. Otherwise, register RT is set to -1.

Special Registers Altered:

None

3.3.20 Prefixed No-Operation Instruction

Prefixed Nop MRR:-form*

pnop

Prefix:

1	3	0	///	0
0	6	8	12	14 31

Suffix:

any value*
0 31

* Value must not correspond to a *Branch* instruction, an ***rfebb*** instruction, a context synchronizing instruction other than ***isync***, or a “Service Processor Attention” instruction

No operation is performed.

Special Registers Altered:

None

Programming Note

The ***pnop*** instruction behaves as a ***b*** $+8 instruction regardless of its suffix. However, it does not cause any side effects such as modification of the Come From Address Register. (see Section 10.2 of Book III).

Programming Note

If the value in the suffix of a ***pnop*** instruction corresponds to a *Branch* instruction, an ***rfebb*** instruction, a context synchronizing instruction other than ***isync***, or a “*Service Processor Attention*” instruction, the instruction form is invalid. The behavior associated with invalid form instructions is described in Section 1.8.2 on page 25. ***rfebb*** and ***isync*** are defined in Book 2: Power ISA Virtual Environment Architecture. Context synchronization and other context synchronizing instructions are defined in Book 3: Power ISA Operating Environment Architecture. *Service Processor Attention* is a reserved instruction; see Appendix C, “Reserved Instructions” on page 1288.

This restriction eases hardware implementation complexity.

Engineering Note

Because the list of word instructions that must not be used as the suffix of ***pnop*** may change in the future, hardware should treat these invalid instruction forms of ***pnop*** either as a no-op or as an illegal instruction. This treatment enhances software compatibility. The choice may vary according to which of the word instructions is used as the suffix.

Chapter 4. Floating-Point Facility

4.1 Floating-Point Facility Overview

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

The processor (augmented by appropriate software support, where required) implements a floating-point system compliant with the ANSI/IEEE Standard 754-1985, "IEEE Standard for Binary Floating-Point Arithmetic" (hereafter referred to as "the IEEE standard"). That standard defines certain required "operations" (addition, subtraction, etc.). Herein, the term "floating-point operation" is used to refer to one of these required operations and to additional operations defined (e.g., those performed by *Multiply-Add* or *Reciprocal Estimate* instructions). A Non-IEEE mode is also provided. This mode, which may produce results not in strict compliance with the IEEE standard, allows shorter latency.

Instructions are provided to perform arithmetic, rounding, conversion, comparison, and other operations in floating-point registers; to move floating-point data between storage and these registers; and to manipulate the Floating-Point Status and Control Register explicitly.

[]

Computational instructions are defined to be 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.

[]

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 $2^{exponent}$. 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. 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 (VXISI)
  - Infinity÷Infinity (VXIDI)
  - 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 134 for a description of these exception and enable bits, and Section 4.4, "Floating-Point Exceptions" on page 142 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 4.1 on page 134.

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.

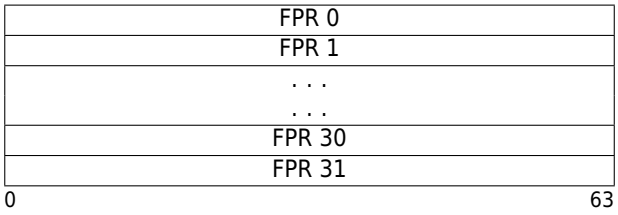


Figure 4.1: Floating-Point Registers

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***, ***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 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.

Figure 4.2: Floating-Point Status and Control Register

The bit definitions for the FPSCR are as follows.

Bit(s) Definition

0:28 Reserved

29:31 **Decimal Rounding Mode** (DRN)

See Section 5.2.1, "DFP Usage of Floating-Point Registers" on page 187.

32 **Floating-Point Exception Summary** (FX)

Every floating-point instruction, except ***mtfsfi*** and ***mtfsf***, implicitly sets FX to 1 if that instruction causes any of the floating-point exception bits in the FPSCR to change from 0 to 1. ***mcrfs***, ***mtfsfi***, ***mtfsf***, ***mtfsb0***, and ***mtfsb1*** can alter FX explicitly.

Programming Note

FX is defined not to be altered implicitly by ***mtfsfi*** and ***mtfsf*** because permitting these instructions to alter FX implicitly could cause a paradox. An example is an ***mtfsfi*** or ***mtfsf*** instruction that supplies 0 for FX and 1 for OX, and is executed when OX=0. See also the Programming Notes with the definition of these two instructions.

33 **Floating-Point Enabled Exception Summary** (FEX)

This bit is the OR of all the floating-point exception bits masked by their respective enable bits. ***mcrfs***, ***mtfsfi***, ***mtfsf***, ***mtfsb0***, and ***mtfsb1*** cannot alter FEX explicitly.

34 **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 VX explicitly.

35 **Floating-Point Overflow Exception** (OX)

See Section 4.4.3, "Overflow Exception" on page 145.

36 **Floating-Point Underflow Exception** (UX)

See Section 4.4.4, "Underflow Exception" on page 145.

37 **Floating-Point Zero Divide Exception** (ZX)

See Section 4.4.2, "Zero Divide Exception" on page 145.

38 **Floating-Point Inexact Exception** (XX)

See Section 4.4.5, “Inexact Exception” on page 146.

XX is a sticky version of FI (see below). Thus the following rules completely describe how XX is set by a given instruction.

- If the instruction affects FI, the new value of XX is obtained by ORing the old value of XX with the new value of FI.
- If the instruction does not affect FI, the value of XX is unchanged.

39 **Floating-Point Invalid Operation Exception (SNaN)** (VXSNAN)

See Section 4.4.1, “Invalid Operation Exception” on page 144.

40 **Floating-Point Invalid Operation Exception ($\infty - \infty$)** (VXISI)

See Section 4.4.1.

41 **Floating-Point Invalid Operation Exception ($\infty \div \infty$)** (VXIDI)

See Section 4.4.1.

42 **Floating-Point Invalid Operation Exception ($0 \div 0$)** (VXZDZ)

See Section 4.4.1.

43 **Floating-Point Invalid Operation Exception ($\infty \times 0$)** (VXIMZ)

See Section 4.4.1.

44 **Floating-Point Invalid Operation Exception (Invalid Compare)** (VXVC)

See Section 4.4.1.

45 **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 140. This bit is not sticky.

46 **Floating-Point Fraction Inexact** (FI)

The last *Arithmetic* or *Rounding and Conversion* instruction either produced an inexact result during rounding or caused a disabled Overflow Exception. See Section 4.3.6. This bit is not sticky.

See the definition of XX, above, regarding the relationship between FI and XX.

47:51 **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.

47 **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 4.3 on page 136.

48:51 **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 4.3 on page 136. Note that in this case the high-order three bits of the FPCC retain their relational significance indicating that the value is less than, greater than, or equal to zero.

48 **Floating-Point Less Than or Negative** (FL or $<$)

49 **Floating-Point Greater Than or Positive** (FG or $>$)

50 **Floating-Point Equal or Zero** (FE or =)

51 **Floating-Point Unordered or NaN** (FU or ?)

52 Reserved

53 **Floating-Point Invalid Operation Exception (Software-Defined Condition)** (VXSOFT)

This bit can be altered only by ***mcrfs***, ***mtfsfi***, ***mtfsf***, ***mtfsb0***, or ***mtfsb1***. See Section 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.

54 **Floating-Point Invalid Operation Exception (Invalid Square Root)** (VXSQRT)

See Section 4.4.1.

55 **Floating-Point Invalid Operation Exception (Invalid Integer Convert)** (VXCVI)

See Section 4.4.1.

56 **Floating-Point Invalid Operation Exception Enable** (VE)

See Section 4.4.1.

57 **Floating-Point Overflow Exception Enable** (OE)

See Section 4.4.3, “Overflow Exception” on page 145.

58 **Floating-Point Underflow Exception Enable** (UE)

See Section 4.4.4, “Underflow Exception” on page 145.

59 **Floating-Point Zero Divide Exception Enable** (ZE)

See Section 4.4.2, “Zero Divide Exception” on page 145.

60 **Floating-Point Inexact Exception Enable** (XE)

See Section 4.4.5, “Inexact Exception” on page 146.

61 **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.

62:63 **Floating-Point Rounding Control** (RN)

See Section 4.3.6, “Rounding” on page 140.

00	Round to Nearest
01	Round toward Zero
10	Round toward +Infinity
11	Round toward -Infinity

Result Flags					Result Value Class
C	<	>	=	?	
1	0	0	0	1	Quiet NaN
0	1	0	0	1	− Infinity
0	1	0	0	0	− Normalized Number
1	1	0	0	0	Denormalized Number
1	0	0	1	0	− Zero
0	0	0	1	0	+ Zero
1	0	1	0	0	+ Denormalized Number
0	0	1	0	0	+ Normalized Number
0	0	1	0	1	+ Infinity

Figure 4.3: Floating-Point Result Flags

4.3 Floating-Point Data

4.3.1 Data Format

This architecture defines the representation of a floating-point value in two different binary fixed-length formats. The format may be a 32-bit single format for a single-precision value or a 64-bit double format for a double-precision value. The single format may be used for data in storage. The double format may be used for data in storage and for data in floating-point registers.

The lengths of the exponent and the fraction fields differ between these two formats. The structure of the single and double formats is shown below.

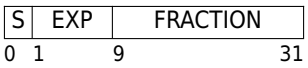


Figure 4.4: Floating-point single format

Figure 4.5: Floating-point double format

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 4.6.

	Format	
	Single	**Double**
Exponent Bias	+127	+1023
Maximum Exponent	+127	+1023
Minimum Exponent	−126	−1022
Widths (bits)		
Format	32	64
Sign	1	1
Exponent	8	11
Fraction	23	52
Significand	24	53

Figure 4.6: IEEE floating-point fields

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 4.7.

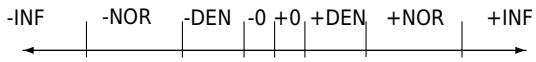


Figure 4.7: Approximation to real numbers

The NaNs are not related to the numeric values or infinities by order or value but are encodings used to convey diagnostic information such as the representation of uninitialized variables.

The following is a description of the different floating-point values defined in the architecture:

Binary floating-point numbers
Machine representable values used as approximations to real numbers. Three categories of numbers are supported: normalized numbers, denormalized numbers, and zero values.

Normalized numbers (± NOR)
These are values that have a biased exponent value in the range:

1 to 254 in single format
1 to 2046 in double format

They are values in which the implied unit bit is 1. Normalized numbers are interpreted as follows:

$$NOR = (-1)^s \times 2^E \times (1.fraction)$$

where s is the sign, E is the unbiased exponent, and 1.fraction is the significand, which is composed of a leading unit bit (implied bit) and a fraction part.

The ranges covered by the magnitude (M) of a normalized floating-point number are approximately equal to:

Single Format:

$$1.2\times10^{-38} \leq M \leq 3.4\times10^{38}$$

Double Format:

$$2.2\times10^{-308} \leq M \leq 1.8\times10^{308}$$

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:

$$DEN = (-1)^s \times 2^{Emin} \times (0.fraction)$$

where 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 144.

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 (VE=0). Quiet NaNs propagate through all floating-point operations except ordered comparison, *Floating Round to Single-Precision*, and conversion to integer. Quiet NaNs do not signal exceptions, except for ordered comparison and conversion to integer operations. Specific encodings in QNaNs can thus be preserved through a sequence of floating-point operations, and used to convey diagnostic information to help identify results from invalid operations.

When a QNaN is the result of a floating-point operation because one of the operands is a NaN or because a QNaN was generated due to a disabled Invalid Operation Exception, then the following rule is applied to determine the NaN with the high-order fraction bit set to 1 that is to be stored as the result.

```
if (FRA) is a NaN
   then FRT ← (FRA)
   else if (FRB) is a NaN
      then if instruction is frsp
           then FRT ← (FRB)0:34 || 29 0
           else FRT ← (FRB)
      else if (FRC) is a NaN
           then FRT ← (FRC)
           else if generated QNaN
                then FRT ← 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 Exception generates this QNaN (i.e., 0x7FF8_0000_0000_0000).

A double-precision NaN is considered to be representable in single format if and only if the low-order 29 bits of the double-precision NaN's fraction are zero.

4.3.3 Sign of Result

The following rules govern the sign of the result of an arithmetic, rounding, or conversion operation, when the operation does not yield an exception. They apply even when the operands or results are zeros or infinities.

- The sign of the result of an add operation is the sign of the operand having the larger absolute value. If both operands have the same sign, the sign of the result of an add operation is the same as the sign of the operands. The sign of the result of the subtract operation $x - y$ is the same as the sign of the result of the add operation $x + (-y)$.
  When the sum of two operands with opposite sign, or the difference of two operands with the same sign, is exactly zero, the sign of the result is positive in all rounding modes except Round toward $-$Infinity, in which mode the sign is negative.
- The sign of the result of a multiply or divide operation is the Exclusive OR of the signs of the operands.
- The sign of the result of a *Square Root* or *Reciprocal Square Root Estimate* operation is always positive, except that the square root of -0 is -0 and the reciprocal square root of -0 is $-$Infinity.
- The sign of the result of a *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 147) 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 145) and Underflow Exception is signaled.

Engineering Note

When denormalized numbers are operands of multiply, divide, and square root operations, some implementations may prenormalize the operands internally before performing the operations.

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.

RN	Rounding Mode
00	Round to Nearest
01	Round toward Zero
10	Round toward +Infinity
11	Round toward -Infinity

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 4.8 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”.

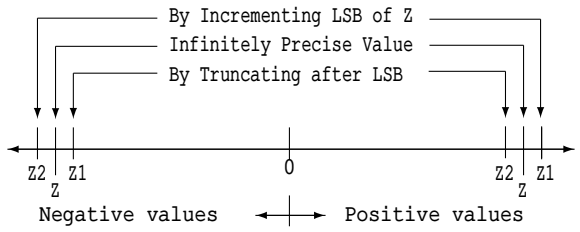


Figure 4.8: Selection of Z1 and Z2

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 147 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 VXSOFT to 1.

Each floating-point exception, and each category of Invalid Operation Exception, has an exception bit in the FPSCR. In addition, each floating-point exception has a corresponding enable bit in the FPSCR. The exception bit indicates occurrence of the corresponding exception. If an exception occurs, the corresponding enable bit governs the result produced by the instruction and, in conjunction with the FE0 and FE1 bits (see page 143), whether and how the system floating-point enabled exception error handler is invoked. (In general, the enabling specified by the enable bit is of invoking the system error handler, not of permitting the exception to occur. The occurrence of an exception depends only on the instruction and its inputs, not on the setting of any control bits. The only deviation from this general rule is that the occurrence of an Underflow Exception may depend on the setting of the enable bit.)

A single instruction, other than ***mtfsfi*** or ***mtfsf***, may set more than one exception bit only in the following cases:

- Inexact Exception may be set with Overflow Exception.
- Inexact Exception may be set with Underflow Exception.
- Invalid Operation Exception (SNaN) is set with Invalid Operation Exception ($\infty \times 0$) 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 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
0	0	**Ignore Exceptions Mode** Floating-point exceptions do not cause the system floating-point enabled exception error handler to be invoked.
0	1	**Imprecise Nonrecoverable Mode** 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.
1	0	**Imprecise Recoverable Mode** 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.
1	1	**Precise Mode** The system floating-point enabled exception error handler is invoked precisely at the instruction that caused the enabled exception.

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.

Engineering Note

It is permissible for the implementation to be precise in any of the three modes that permit interrupts, or to be recoverable in Nonrecoverable Mode.

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 ***mtfsb1*** instruction is executed that sets VXSOFT to 1 (Software-Defined Condition).

4.4.1.2 Action

The action to be taken depends on the setting of the Invalid Operation Exception Enable bit of the FPSCR.

When Invalid Operation Exception is enabled (VE=1) and an Invalid Operation Exception occurs, the following actions are taken:

1. One or two Invalid Operation Exceptions are set

VXSNAN	(if SNaN)
VXISI	(if $\infty - \infty$)
VXIDI	(if $\infty \div \infty$)
VXZDZ	(if $0 \div 0$)
VXIMZ	(if $\infty \times 0$)
VXVC	(if invalid compare)
VXSOFT	(if software-defined condition)
VXSQRT	(if invalid square root)
VXCVI	(if invalid integer convert)

2. If the operation is an arithmetic, *Floating Round to Single-Precision*, *Floating Round to Integer,* or convert to integer operation,
   the target FPR is unchanged
   FR FI are set to zero
   FPRF is unchanged
3. If the operation is a compare,
   FR FI C are unchanged
   FPCC is set to reflect unordered
4. If an ***mtfsfi***, ***mtfsf***, or ***mtfsb1*** instruction is executed that sets VXSOFT to 1,
   The FPSCR is set as specified in the instruction description.

When Invalid Operation Exception is disabled (VE=0) and an Invalid Operation Exception occurs, the following actions are taken:

1. One or two Invalid Operation Exceptions are set

VXSNAN	(if SNaN)
VXISI	(if $\infty - \infty$)
VXIDI	(if $\infty \div \infty$)
VXZDZ	(if $0 \div 0$)
VXIMZ	(if $\infty \times 0$)
VXVC	(if invalid compare)
VXSOFT	(if software-defined condition)
VXSQRT	(if invalid square root)
VXCVI	(if invalid integer convert)

2. If the operation is an arithmetic or *Floating Round to Single-Precision* operation,
   the target FPR is set to a Quiet NaN
   FR FI are set to zero
   FPRF is set to indicate the class of the result (Quiet NaN)
3. If the operation is a convert to 64-bit integer operation,
   the target FPR is set as follows:
   FRT is set to the most positive 64-bit integer if the operand in FRB is a positive number or $+\infty$, and to the most negative 64-bit integer if the operand in FRB is a negative number, $-\infty$, or NaN
   FR FI are set to zero
   FPRF is undefined
4. If the operation is a convert to 32-bit integer operation,
   the target FPR is set as follows:
   $FRT_{0:31} \leftarrow$ undefined
   $FRT_{32:63}$ are set to the most positive 32-bit integer if the operand in FRB is a positive number or +infinity, and to the most negative 32-bit integer if the operand in FRB is a negative number, -infinity, or NaN
   FR FI are set to zero
   FPRF is undefined
5. If the operation is a compare,
   FR FI C are unchanged
   FPCC is set to reflect unordered
6. If an ***mtfsfi***, ***mtfsf***, or ***mtfsb1*** instruction is executed that sets VXSOFT to 1,
   The FPSCR is set as specified in the instruction description.

4.4.2 Zero Divide Exception

4.4.2.1 Definition

A Zero Divide Exception occurs when a *Divide* instruction is executed with a zero divisor value and a finite nonzero dividend value. It also occurs when a *Reciprocal Estimate* instruction (***fre***[***s***] or ***frsqrte***[***s***]) is executed with an operand value of zero.

Architecture Note

The name is a misnomer used for historical reasons. The proper name for this exception should be "Exact Infinite Result from Finite Operands" corresponding to what mathematicians call a "pole".

4.4.2.2 Action

The action to be taken depends on the setting of the Zero Divide Exception Enable bit of the FPSCR.

When Zero Divide Exception is enabled (ZE=1) and a Zero Divide Exception occurs, the following actions are taken:

1. Zero Divide Exception is set
   ZX ← 1
2. The target FPR is unchanged
3. FR FI are set to zero
4. FPRF is unchanged

When Zero Divide Exception is disabled (ZE=0) and a Zero Divide Exception occurs, the following actions are taken:

1. Zero Divide Exception is set
   ZX ← 1
2. The target FPR is set to ± Infinity, where the sign is determined by the XOR of the signs of the operands
3. FR FI are set to zero
4. FPRF is set to indicate the class and sign of the result (± Infinity)

4.4.3 Overflow Exception

4.4.3.1 Definition

An Overflow Exception occurs when the magnitude of what would have been the rounded result if the exponent range were unbounded exceeds that of the largest finite number of the specified result precision.

4.4.3.2 Action

The action to be taken depends on the setting of the Overflow Exception Enable bit of the FPSCR.

When Overflow Exception is enabled (OE=1) and an Overflow Exception occurs, the following actions are taken:

1. Overflow Exception is set
   OX ← 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. FPRF is set to indicate the class and sign of the result (± Normal Number)

When Overflow Exception is disabled (OE=0) and an Overflow Exception occurs, the following actions are taken:

1. Overflow Exception is set
   OX ← 1
2. Inexact Exception is set
   XX ← 1
3. The result is determined by the rounding mode (RN) and the sign of the intermediate result as follows:
   - Round to Nearest
     Store ± Infinity, where the sign is the sign of the intermediate result
   - Round toward Zero
     Store the format's largest finite number with the sign of the intermediate result
   - Round toward + Infinity
     For negative overflow, store the format's most negative finite number; for positive overflow, store +Infinity
   - Round toward − Infinity
     For negative overflow, store −Infinity; for positive overflow, store the format's largest finite number
4. The result is placed into the target FPR
5. FR is undefined
6. FI is set to 1
7. FPRF is set to indicate the class and sign of the result (± Infinity or ± 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 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) then the intermediate result is denormalized (see Section 4.3.4, “Normalization and Denormalization” on page 138) and rounded (see Section 4.3.6, “Rounding” on page 140) before being placed into the target FPR.

“Loss of Accuracy” is detected when the delivered result value differs from what would have been computed were both the precision and the exponent range unbounded.

4.4.4.2 Action

The action to be taken depends on the setting of the Underflow Exception Enable bit of the FPSCR.

When Underflow Exception is enabled (UE=1) and an Underflow Exception occurs, the following actions are taken:

1. Underflow Exception is set
   UX ← 1
2. For double-precision arithmetic instructions, the exponent of the normalized intermediate result is adjusted by adding 1536
3. For single-precision arithmetic instructions and the *Floating Round to Single-Precision* instruction, the exponent of the normalized intermediate result is adjusted by adding 192
4. The adjusted rounded result is placed into the target FPR
5. FPRF is set to indicate the class and sign of the result (± Normalized Number)

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.

When Underflow Exception is disabled (UE=0) and an Underflow Exception occurs, the following actions are taken:

1. Underflow Exception is set
   UX ← 1
2. The rounded result is placed into the target FPR
3. FPRF is set to indicate the class and sign of the result (± Normalized Number, ± Denormalized Number, or ± Zero)

4.4.5 Inexact Exception

4.4.5.1 Definition

An Inexact Exception occurs when one of two conditions occur during rounding:

1. The rounded result differs from the intermediate result assuming both the precision and the exponent range of the intermediate result to be unbounded. In this case the result is said to be inexact. (If the rounding causes an enabled Overflow Exception or an enabled Underflow Exception, an Inexact Exception also occurs only if the significands of the rounded result and the intermediate result differ.)
2. The rounded result overflows and Overflow Exception is disabled.

4.4.5.2 Action

The action to be taken does not depend on the setting of the Inexact Exception Enable bit of the FPSCR.

When an Inexact Exception occurs, the following actions are taken:

1. Inexact Exception is set
   XX ← 1
2. The rounded or overflowed result is placed into the target FPR
3. FPRF is set to indicate the class and sign of the result

Programming Note

In some implementations, enabling Inexact Exceptions may degrade performance more than does enabling other types of floating-point exception.

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 4.9: IEEE 64-bit execution model

The S bit is the sign bit.

The C bit is the carry bit, which captures the carry out of the significand.

The L bit is the leading unit bit of the significand, which receives the implicit bit from the operand.

The 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 4.10 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).

G R X	Interpretation
0 0 0	IR is exact
0 0 1	IR closer to NL
0 1 0	
0 1 1	
1 0 0	IR midway between NL and NH
1 0 1	IR closer to NH
1 1 0	
1 1 1	

Figure 4.10: Interpretation of G, R, and X bits

Figure 4.11 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 4.9.

Format	Guard	Round	Sticky
Double	G bit	R bit	X bit
Single	24	25	OR of 26:52, G, R, X

Figure 4.11: Location of the Guard, Round, and Sticky bits in the IEEE execution model

The significand of the intermediate result is prepared for rounding by shifting its contents right, if required, until the least significant bit to be retained is in the low-order bit position of the fraction. Four user-selectable rounding modes are provided through RN as described in Section 4.3.6, “Rounding” on page 140. Using Z1 and Z2 as defined on page 140, 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 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.

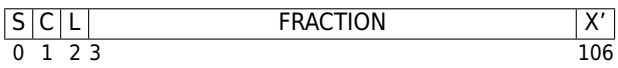


Figure 4.12: Multiply-add 64-bit execution model

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), 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 4.13 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.

Format	Guard	Round	Sticky
Double	53	54	OR of 55:105, X′
Single	24	25	OR of 26:105, X′

Figure 4.13: 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 4.5.1.

If the instruction is *Floating Negative Multiply-Add* or *Floating Negative Multiply-Subtract*, the final result is negated.

4.6 Floating-Point Facility Instructions

4.6.1 Floating-Point Storage Access Instructions

The *Storage Access* instructions compute the effective address (EA) of the storage to be accessed as described in Section 1.10.3, “Effective Address Calculation” on page 31.

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 Mnemonics” on page 965.

4.6.1.1 Storage Access Exceptions

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

4.6.2 Floating-Point Load Instructions

There are three basic forms of load instruction: single-precision, double-precision, and integer. The integer form is provided by the *Load Floating-Point as Integer Word Algebraic* instruction, described on page 153. Because the FPRs support only floating-point double format, single-precision *Load Floating-Point* instructions convert single-precision data to double format prior to loading the operand into the target FPR. The conversion and loading steps are as follows.

Let $\mathtt{WORD}_{0:31}$ be the floating-point single-precision operand accessed from storage.

Normalized Operand

```
if WORD1:8 > 0 and WORD1:8 < 255 then
   FRT0:1 ← WORD0:1
   FRT2 ← ¬WORD1
   FRT3 ← ¬WORD1
   FRT4 ← ¬WORD1
   FRT5:63 ← WORD2:31 || 29 0
```

Denormalized Operand

```
if WORD1:8 = 0 and WORD9:31 ≠ 0 then
sign ← WORD0
exp ← -126
frac0:52 ← 0b0 || WORD9:31 || 29 0
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 || 29 0
```

Engineering Note

The above description of the conversion steps is a model only. The actual implementation may vary from this but must produce results equivalent to what this model would produce.

For double-precision *Load Floating-Point* instructions and for the *Load Floating-Point as Integer Word Algebraic* instruction no conversion is required, as the data from storage are copied directly into the FPR.

Many of the *Load Floating-Point* instructions have an “update” form, in which register RA is updated with the effective address. For these forms, if RA≠0, the effective address is placed into register RA and the storage element (word or doubleword) addressed by EA is loaded into FRT.

Note: Recall that RA and RB denote General Purpose Registers, while FRT denotes a Floating-Point Register.

Load Floating-Point Single D-form

lfs FRT,D(RA)

48	FRT	RA	D
0	6	11	16 31

Prefixed Load Floating-Point Single MLS:D-form

plfs FRT,D(RA),R

Prefix:

1	2	0	//	R	//	d0
0	6	8	9	11	12	14 31

Suffix:

48	FRT	RA	d1
0	6	11	16 31

```
if ''lfs''  then
   EA ← (RA|0) + EXTS64(D)
if ''plfs'' & R=0 then
   EA ← (RA|0) + EXTS64(d0||d1)
if ''plfs'' & R=1 then
   EA ← CIA + EXTS64(d0||d1)

FRT ← DOUBLE(MEM(EA, 4))
```

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

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

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

The word in storage addressed by EA is interpreted as a floating-point single-precision operand. This word is converted to floating-point double format (see page 149) and placed into register FRT.

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

Special Registers Altered:

None

Extended Mnemonics:

Extended Mnemonics for *Prefixed Load Floating-Point Single*:

Extended mnemonic:		Equivalent to:	
plfs	FRT,D(RA)	plfs	FRT,D(RA),0
plfs	FRT,D	plfs	FRT,D(0),1

Load Floating-Point Single Indexed X-form

lfsx FRT,RA,RB

31	FRT	RA	RB	535	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + (RB)
FRT ← DOUBLE(MEM(EA, 4))
```

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

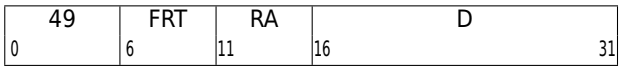
The word in storage addressed by EA is interpreted as a floating-point single-precision operand. This word is converted to floating-point double format (see page 149) and placed into register FRT.

Special Registers Altered:

None

Load Floating-Point Single with Update D-form

lfsu FRT,D(RA)




```
EA ← (RA) + EXTS(D)
FRT ← DOUBLE(MEM(EA, 4))
RA ← EA
```

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

The word in storage addressed by EA is interpreted as a floating-point single-precision operand. This word is converted to floating-point double format (see page 149) and placed into register FRT.

EA is placed into register RA.

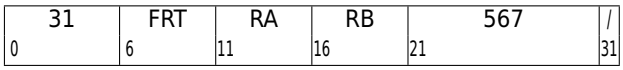
If RA=0, the instruction form is invalid.

Special Registers Altered:

None

Load Floating-Point Single with Update Indexed X-form

lfsux FRT,RA,RB




```
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 149) and placed into register FRT.

EA is placed into register RA.

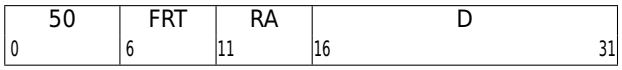
If RA=0, the instruction form is invalid.

Special Registers Altered:

None

Load Floating-Point Double D-form

lfd FRT,D(RA)

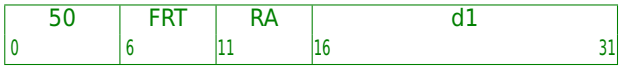


Prefixed Load Floating-Point Double MLS:D-form

plfd FRT,D(RA),R

Prefix:

Suffix:




```
if ''lfd''  then
   EA ← (RA|0) + EXTS64(D)
if ''plfd'' & R=0 then
   EA ← (RA|0) + EXTS64(d0||d1)
if ''plfd'' & R=1 then
   EA ← CIA + EXTS64(d0||d1)

FRT ← MEM(EA, 8)
```

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

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

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

The doubleword in storage addressed by EA is loaded into register FRT.

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

Special Registers Altered:

None

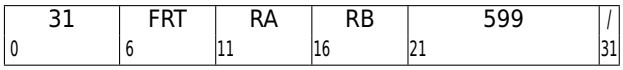
Extended Mnemonics:

Extended Mnemonics for *Prefixed Load Floating-Point Double*:

Extended mnemonic:		Equivalent to:	
plfd	FRT,D(RA)	plfd	FRT,D(RA),0
plfd	FRT,D	plfd	FRT,D(0),1

Load Floating-Point Double Indexed X-form

lfdx FRT,RA,RB




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

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

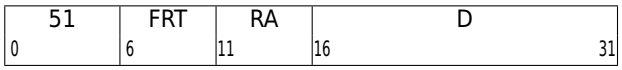
The doubleword in storage addressed by EA is loaded into register FRT.

Special Registers Altered:

None

Load Floating-Point Double with Update D-form

lfdu FRT,D(RA)




```
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.

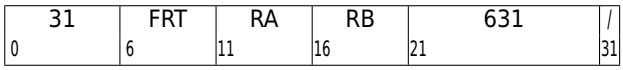
If RA=0, the instruction form is invalid.

Special Registers Altered:

None

Load Floating-Point Double with Update Indexed X-form

lfdux FRT,RA,RB




```
EA ← (RA) + (RB)
FRT ← MEM(EA, 8)
RA ← EA
```

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

The doubleword in storage addressed by EA is loaded into register FRT.

EA is placed into register RA.

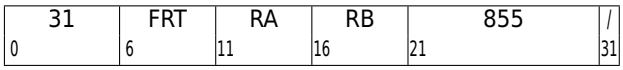
If RA=0, the instruction form is invalid.

Special Registers Altered:

None

Load Floating-Point as Integer Word Algebraic Indexed X-form

lfiwax FRT,RA,RB




```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + (RB)
FRT ← EXTS(MEM(EA, 4))
```

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

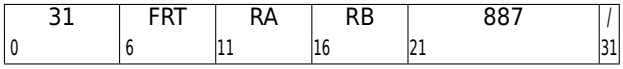
The word in storage addressed by EA is loaded into $FRT_{32:63}$. $FRT_{0:31}$ are filled with a copy of bit 0 of the loaded word.

Special Registers Altered:

None

Load Floating-Point as Integer Word & Zero Indexed X-form

lfiwzx FRT,RA,RB




```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + (RB)
FRT ← 32 0 || 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 157. 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 $WORD_{0:31}$ be the word in storage written to.

No Denormalization Required (includes Zero / Infinity / NaN)

```
if FRS1:11 > 896 or FRS1:63 = 0 then
   WORD0:1 ← FRS0:1
   WORD2:31 ← FRS5:34
```

Denormalization Required

```
if 874 ≤ FRS1:11 ≤ 896 then
   sign ← FRS0
   exp ← FRS1:11 - 1023
   frac0:52 ← 0b1 || FRS12:63
   denormalize operand
      do while exp < -126
         frac0:52 ← 0b0 || frac0:51
         exp ← exp + 1
   WORD0 ← sign
   WORD1:8 ← 0x00
   WORD9:31 ← frac1:23
else WORD ← undefined
```

Notice that if the value to be stored by a single-precision *Store Floating-Point* instruction is larger in magnitude than the maximum number representable in single format, the first case above (No Denormalization Required) applies. The result stored in WORD is then a well-defined value, but is not numerically equal to the value in the source register (i.e., the result of a single-precision *Load Floating-Point* from WORD will not compare equal to the contents of the original source register).

Engineering Note

The above description of the conversion steps is a model only. The actual implementation may vary from this but must produce results equivalent to what this model would produce.

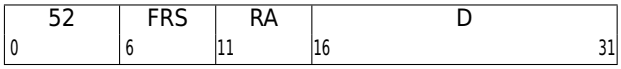
For double-precision *Store Floating-Point* instructions and for the *Store Floating-Point as Integer Word* instruction no conversion is required, as the data from the FPR are copied directly into storage.

Many of the *Store Floating-Point* instructions have an "update" form, in which register RA is updated with the effective address. For these forms, if RA≠0, the effective address is placed into register RA.

Note: Recall that RA and RB denote General Purpose Registers, while FRS denotes a Floating-Point Register.

Store Floating-Point Single D-form

stfs FRS,D(RA)

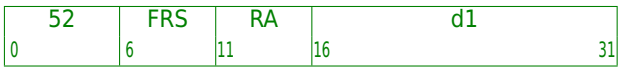


Prefixed Store Floating-Point Single MLS:D-form

pstfs FRS,D(RA),R

Prefix:

Suffix:




```
if ''stfs''  then
   EA ← (RA|0) + EXTS64(D)
if ''pstfs'' & R=0 then
   EA ← (RA|0) + EXTS64(d0||d1)
if ''pstfs'' & R=1 then
   EA ← CIA + EXTS64(d0||d1)

MEM(EA, 4) ← SINGLE((FRS))
```

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

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

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

The contents of register FRS are converted to single format (see page 154) and stored into the word in storage addressed by EA.

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

Special Registers Altered:

None

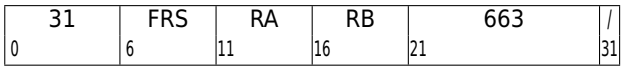
Extended Mnemonics:

Extended Mnemonics for *Prefixed Store Floating-Point Single*:

Extended mnemonic:		Equivalent to:	
pstfs	FRS,D(RA)	pstfs	FRS,D(RA),0
pstfs	FRS,D	pstfs	FRS,D(0),1

Store Floating-Point Single Indexed X-form

stfsx FRS,RA,RB




```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + (RB)
MEM(EA, 4) ← SINGLE((FRS))
```

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

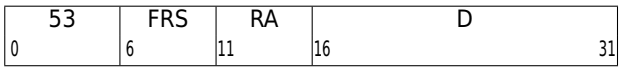
The contents of register FRS are converted to single format (see page 154) and stored into the word in storage addressed by EA.

Special Registers Altered:

None

Store Floating-Point Single with Update D-form

stfsu FRS,D(RA)




```
EA ← (RA) + EXTS(D)
MEM(EA, 4) ← SINGLE((FRS))
RA ← EA
```

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

The contents of register FRS are converted to single format (see page 154) and stored into the word in storage addressed by EA.

EA is placed into register RA.

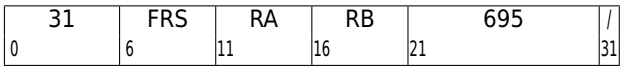
If RA=0, the instruction form is invalid.

Special Registers Altered:

None

Store Floating-Point Single with Update Indexed X-form

stfsux FRS,RA,RB




```
EA ← (RA) + (RB)
MEM(EA, 4) ← SINGLE((FRS))
RA ← EA
```

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

The contents of register FRS are converted to single format (see page 154) and stored into the word in storage addressed by EA.

EA is placed into register RA.

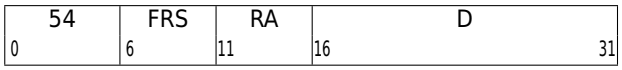
If RA=0, the instruction form is invalid.

Special Registers Altered:

None

Store Floating-Point Double D-form

stfd FRS,D(RA)

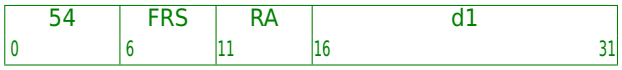


Prefixed Store Floating-Point Double MLS:D-form

pstfd FRS,D(RA),R

Prefix:

Suffix:




```
if ''stfd''  then
   EA ← (RA|0) + EXTS64(D)
if ''pstfd'' & R=0 then
   EA ← (RA|0) + EXTS64(d0||d1)
if ''pstfd'' & R=1 then
   EA ← CIA + EXTS64(d0||d1)

MEM(EA, 8) ← (FRS)
```

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

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

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

The contents of register FRS are stored into the double-word in storage addressed by EA.

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

Special Registers Altered:

None

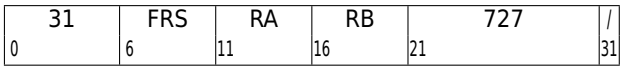
Extended Mnemonics:

Extended Mnemonics for *Prefixed Store Floating-Point Double*:

Extended mnemonic:	Equivalent to:
pstfd FRS,D(RA)	pstfd FRS,D(RA),0
pstfd FRS,D	pstfd FRS,D(0),1

Store Floating-Point Double Indexed X-form

stfdx FRS,RA,RB




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

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

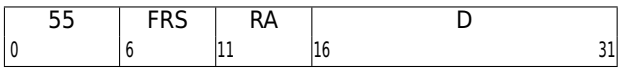
The contents of register FRS are stored into the double-word in storage addressed by EA.

Special Registers Altered:

None

Store Floating-Point Double with Update D-form

stfdu FRS,D(RA)




```
EA ← (RA) + EXTS(D)
MEM(EA, 8) ← (FRS)
RA ← EA
```

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

The contents of register FRS are stored into the double-word in storage addressed by EA.

EA is placed into register RA.

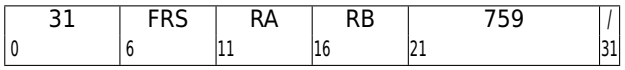
If RA=0, the instruction form is invalid.

Special Registers Altered:

None

Store Floating-Point Double with Update Indexed X-form

stfdux FRS,RA,RB




```
EA ← (RA) + (RB)
MEM(EA, 8) ← (FRS)
RA ← EA
```

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

The contents of register FRS are stored into the double-word in storage addressed by EA.

EA is placed into register RA.

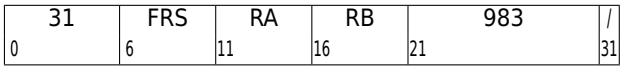
If RA=0, the instruction form is invalid.

Special Registers Altered:

None

Store Floating-Point as Integer Word Indexed X-form

stfiwx FRS,RA,RB




```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + (RB)
MEM(EA, 4) ← (FRS)32:63
```

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

$(FRS)_{32:63}$ are stored, without conversion, into the word in storage addressed by EA.

If the contents of register FRS were produced, either directly or indirectly, by a *Load Floating-Point Single* instruction, a single-precision *Arithmetic* instruction, or ***frsp***, then the value stored is undefined. (The contents of register FRS are produced directly by such an instruction if FRS is the target register for the instruction. The contents of register FRS are produced indirectly by such an instruction if FRS is the final target register of a sequence of one or more *Floating-Point Move* instructions, with the input to the sequence having been produced directly by such an instruction.)

Special Registers Altered:

None

Architecture Note

Allowing the value stored to be undefined if the input to ***stfiwx*** was produced by a single-precision-producing instruction (i.e., a *Load Floating-Point Single* instruction, a single-precision arithmetic instruction, or ***frsp***) seems gratuitous at the architectural level. The background and reasons for allowing it are as follows.

- The implementors agreed to support ***stfiwx*** partly because they understood it to be easy to implement.
- In some implementations (e.g., those that keep single-precision numbers in registers in a non-architected format), storing the architected low-order 32 bits of a register that was set by a single-precision-producing instruction may be harder (and slower, and more trouble to verify) than simply storing whatever happens to be in the low-order 32 bits of the register.
- Software can think of no use for storing the low-order 32 bits of the result of a single-precision producing instruction.

4.6.4 Floating-Point Load and Store Double Pair Instructions [Phased-Out]

For ***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.

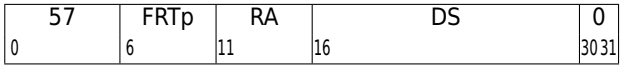
For ***stfdp***[***x***], the content of an even-odd pair of FPRs is stored into the doubleword-pair in storage addressed by EA, with the even-numbered FPR being stored into the leftmost doubleword in storage and the odd-numbered FPR being stored into the rightmost doubleword.

Programming Note

The instructions described in this section should not be used to access an operand in DFP Extended format when the processor is in Little-Endian mode.

Load Floating-Point Double Pair DS-form

lfdp FRTp,disp(RA)




```
if RA = 0 then b ← 0
else          b ←(RA)
EA ← b + EXTS(DS||0b00)
FRTp_even ← MEM(EA,8)
FRTp_odd  ← 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.

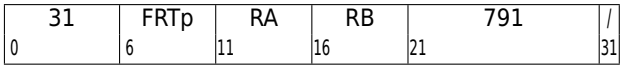
If FRTp is odd, the instruction form is invalid.

Special Registers Altered:

None

Load Floating-Point Double Pair Indexed X-form

lfdpx FRTp,RA,RB




```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + (RB)
FRTp_even ← MEM(EA,8)
FRTp_odd  ← 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.

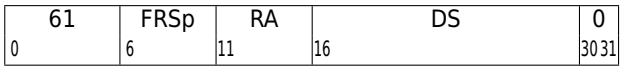
If FRTp is odd, the instruction form is invalid.

Special Registers Altered:

None

Store Floating-Point Double Pair DS-form

stfdp FRSp,disp(RA)




```
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.

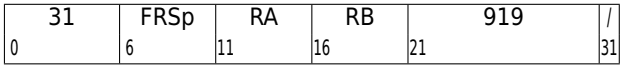
If FRSp is odd, the instruction form is invalid.

Special Registers Altered:

None

Store Floating-Point Double Pair Indexed X-form

stfdpx FRSp,RA,RB




```
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) + (DS||0b00).

The contents of the even-numbered register of FRSp are stored into the doubleword in storage addressed by EA.

The contents of the odd-numbered register of FRSp are stored into the doubleword in storage addressed by EA+8.

If FRSp is odd, the instruction form is invalid.

Special Registers Altered:

None

4.6.5 Floating-Point Move Instructions

These instructions copy data from one floating-point register to another, altering the sign bit (bit 0) as described below for ***fneg***, ***fabs***, ***fnabs***, and ***fcpsgn***. These instructions treat NaNs just like any other kind of value (e.g., the sign bit of a NaN may be altered by ***fneg***, ***fabs***, ***fnabs***, and ***fcpsgn***). These instructions do not alter the FPSCR.

Floating Move Register X-form

fmr	FRT,FRB	(Rc=0)
fmr.	FRT,FRB	(Rc=1)

63	FRT	///	FRB	72	Rc
0	6	11	16	21	31

The contents of register FRB are placed into register FRT.

Special Registers Altered:

Register	*Field(s)*	
CR	CR1	(if Rc=1)

Floating Negate X-form

fneg	FRT,FRB	(Rc=0)
fneg.	FRT,FRB	(Rc=1)

63	FRT	///	FRB	40	Rc
0	6	11	16	21	31

The contents of register FRB with bit 0 inverted are placed into register FRT.

Special Registers Altered:

Register	*Field(s)*	
CR	CR1	(if Rc=1)

Floating Absolute Value X-form

fabs	FRT,FRB	(Rc=0)
fabs.	FRT,FRB	(Rc=1)

63	FRT	///	FRB	264	Rc
0	6	11	16	21	31

The contents of register FRB with bit 0 set to zero are placed into register FRT.

Special Registers Altered:

Register	*Field(s)*	
CR	CR1	(if Rc=1)

Floating Negative Absolute Value X-form

fnabs	FRT,FRB	(Rc=0)
fnabs.	FRT,FRB	(Rc=1)

63	FRT	///	FRB	136	Rc
0	6	11	16	21	31

The contents of register FRB with bit 0 set to one are placed into register FRT.

Special Registers Altered:

Register	*Field(s)*	
CR	CR1	(if Rc=1)

Floating Copy Sign X-form

fcpsgn	FRT,FRA,FRB	(Rc=0)
fcpsgn.	FRT,FRA,FRB	(Rc=1)

63	FRT	FRA	FRB	8	Rc
0	6	11	16	21	31

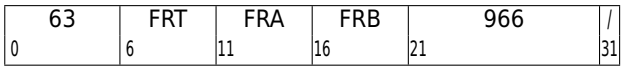
The contents of register FRB with bit 0 set to the value of bit 0 of register FRA are placed into register FRT.

Special Registers Altered:

Register	*Field(s)*	
CR	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]`.

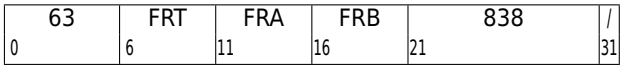
fmrgew is treated as a *Floating-Point* instruction in terms of resource availability.

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]
FPR[FRT].word[1] ← FPR[FRB].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

Programming Note

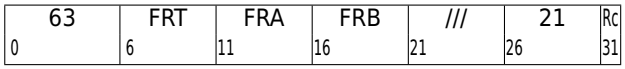
fmrgew and ***fmrgow*** are provided to support direct move operations in 32-bit mode.

4.6.6 Floating-Point Arithmetic Instructions

4.6.6.1 Floating-Point Elementary Arithmetic Instructions

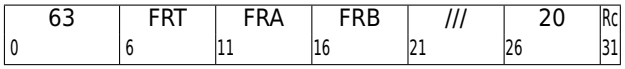
Floating Add A-form

fadd	FRT,FRA,FRB	(Rc=0)
fadd.	FRT,FRA,FRB	(Rc=1)



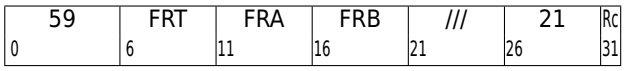
Floating Subtract A-form

fsub	FRT,FRA,FRB	(Rc=0)
fsub.	FRT,FRA,FRB	(Rc=1)



Floating Add Single A-form

fadds	FRT,FRA,FRB	(Rc=0)
fadds.	FRT,FRA,FRB	(Rc=1)



The floating-point operand in register FRA is added to the floating-point operand in register FRB.

If the most significant bit of the resultant significand is not 1, the result is normalized. The result is rounded to the target precision under control of RN and placed into register FRT.

Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate sum. All 53 bits of the significand as well as all three guard bits (G, R, and X) enter into the computation.

If a carry occurs, the sum's significand is shifted right one bit position and the exponent is increased by one.

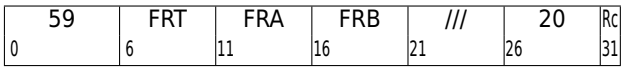
FPRF is set to the class and sign of the result, except for Invalid Operation Exceptions when VE=1.

Special Registers Altered:

Register	*Field(s)*	
FPSCR	FPRF FR FI FX OX UX XX VXSNAN VXISI	
CR	CR1	(if Rc=1)

Floating Subtract Single A-form

fsubs	FRT,FRA,FRB	(Rc=0)
fsubs.	FRT,FRA,FRB	(Rc=1)



The floating-point operand in register FRB is subtracted from the floating-point operand in register FRA.

If the most significant bit of the resultant significand is not 1, the result is normalized. The result is rounded to the target precision under control of RN and placed into register FRT.

The execution of the Floating Subtract instruction is identical to that of Floating Add, except that the contents of FRB participate in the operation with the sign bit (bit 0) inverted.

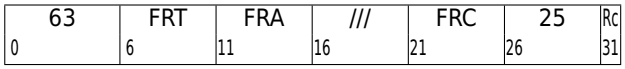
FPRF is set to the class and sign of the result, except for Invalid Operation Exceptions when VE=1.

Special Registers Altered:

Register	*Field(s)*	
FPSCR	FPRF FR FI FX OX UX XX VXSNAN VXISI	
CR	CR1	(if Rc=1)

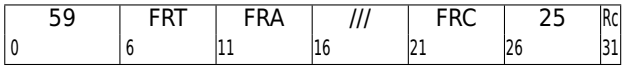
Floating Multiply A-form

fmul	FRT,FRA,FRC	(Rc=0)
fmul.	FRT,FRA,FRC	(Rc=1)



Floating Multiply Single A-form

fmuls	FRT,FRA,FRC	(Rc=0)
fmuls.	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 RN and placed into register FRT.

Floating-point multiplication is based on exponent addition and multiplication of the significands.

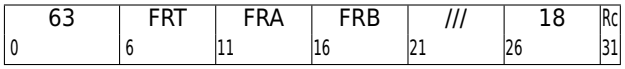
FPRF is set to the class and sign of the result, except for Invalid Operation Exceptions when VE=1.

Special Registers Altered:

Register	*Field(s)*	
FPSCR	FPRF FR FI FX OX UX XX VXSNAN VXIMZ	
CR	CR1	(if Rc=1)

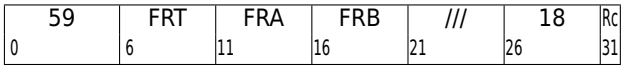
Floating Divide A-form

fdiv	FRT,FRA,FRB	(Rc=0)
fdiv.	FRT,FRA,FRB	(Rc=1)



Floating Divide Single A-form

fdivs	FRT,FRA,FRB	(Rc=0)
fdivs.	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 RN and placed into register FRT.

Floating-point division is based on exponent subtraction and division of the significands.

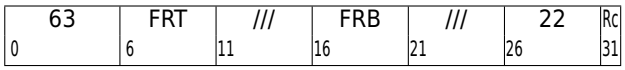
FPRF is set to the class and sign of the result, except for Invalid Operation Exceptions when VE=1 and Zero Divide Exceptions when ZE=1.

Special Registers Altered:

Register	*Field(s)*	
FPSCR	FPRF FR FI FX OX UX ZX XX VXSNAN VXIDI VXZDZ	
CR	CR1	(if Rc=1)

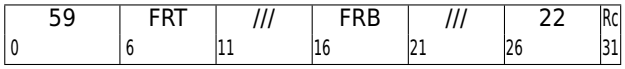
Floating Square Root A-form

fsqrt	FRT,FRB	(Rc=0)
fsqrt.	FRT,FRB	(Rc=1)



Floating Square Root Single A-form

fsqrts	FRT,FRB	(Rc=0)
fsqrts.	FRT,FRB	(Rc=1)



The square root of the floating-point operand in register FRB is placed into register FRT.

If the most significant bit of the resultant significand is not 1, the result is normalized. The result is rounded to the target precision under control of RN and placed into register FRT.

Operation with various special values of the operand is summarized below.

Operand	Result	Exception
$-\infty$	QNaN[1]	VXSQRT
< 0	QNaN[1]	VXSQRT
-0	-0	None
$+\infty$	$+\infty$	None
SNaN	QNaN[1]	VXSNAN
QNaN	QNaN	None

[1] No result if VE=1

$FPSCR_{FPRF}$ is set to the class and sign of the result, except for Invalid Operation Exceptions when $FPSCR_{VE}$=1.

Special Registers Altered:

Register	*Field(s)*	
FPSCR	FPRF FR FI FX OX UX XX VXSNAN VXSQRT	
CR	CR1	(if Rc=1)

Floating Reciprocal Estimate A-form

fre	FRT,FRB	(Rc=0)
fre.	FRT,FRB	(Rc=1)

63	FRT	///	FRB	///	24	Rc
0	6	11	16	21	26	31

Floating Reciprocal Estimate Single A-form

fres	FRT,FRB	(Rc=0)
fres.	FRT,FRB	(Rc=1)

59	FRT	///	FRB	///	24	Rc
0	6	11	16	21	26	31

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.,

$$\left|\frac{\text{estimate} - 1/x}{1/x}\right| \leq \frac{1}{256}$$

where x is the initial value in FRB.

Operation with various special values of the operand is summarized below.

Operand	Result	Exception
$-\infty$	-0	None
-0	$-\infty$[1]	ZX
$+0$	$+\infty$[1]	ZX
$+\infty$	$+0$	None
SNaN	QNaN[2]	VXSNAN
QNaN	QNaN	None

[1] No result if ZE=1.
[2] No result if VE=1.

FPRF is set to the class and sign of the result, except for Invalid Operation Exceptions when VE=1 and Zero Divide Exceptions when ZE=1.

The results of executing this instruction may vary between implementations, and between different executions on the same implementation.

Special Registers Altered:

Register	*Field(s)*	
FPSCR	FPRF FX OX UX ZX VXSNAN	
FPSCR	FR (undefined)	
FPSCR	FI (undefined)	
FPSCR	XX (undefined)	
CR	CR1	(if Rc=1)

Programming Note

For the *Floating-Point Estimate* instructions, some implementations might implement a precision higher than the minimum architected precision. Thus, a program may take advantage of the higher precision instructions to increase performance by decreasing the iterations needed for software emulation of floating-point instructions. However, there is no guarantee given about the precision which may vary (up or down) between implementations. Only programs targeted at a specific implementation (i.e., the program will not be migrated to another implementation) should take advantage of the higher precision of the instructions. All other programs should rely on the minimum architected precision, which will guarantee the program to run properly across different implementations.

Floating Reciprocal Square Root Estimate A-form

frsqrte	FRT,FRB	(Rc=0)
frsqrte.	FRT,FRB	(Rc=1)

63	FRT	///	FRB	///	26	Rc
0	6	11	16	21	26	31

Floating Reciprocal Square Root Estimate Single A-form

frsqrtes	FRT,FRB	(Rc=0)
frsqrtes.	FRT,FRB	(Rc=1)

59	FRT	///	FRB	///	26	Rc
0	6	11	16	21	26	31

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.,

$$\left|\frac{\text{estimate} - 1/(\sqrt{x})}{1/(\sqrt{x})}\right| \leq \frac{1}{32}$$

where x is the initial value in FRB.

Operation with various special values of the operand is summarized below.

Operand	Result	Exception
$-\infty$	QNaN[2]	VXSQRT
< 0	QNaN[2]	VXSQRT
-0	$-\infty$[1]	ZX
$+0$	$+\infty$[1]	ZX
$+\infty$	$+0$	None
SNaN	QNaN[2]	VXSNAN
QNaN	QNaN	None

[1] No result if ZE=1.
[2] No result if VE=1.

FPRF is set to the class and sign of the result, except for Invalid Operation Exceptions when VE=1 and Zero Divide Exceptions when ZE=1.

The results of executing this instruction may vary between implementations, and between different executions on the same implementation.

Special Registers Altered:

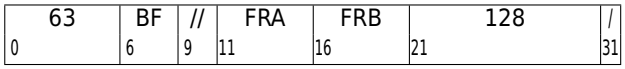
Register	*Field(s)*	
FPSCR	FPRF FX OX UX ZX VXSNAN VXSQRT	
FPSCR	FR (undefined)	
FPSCR	FI (undefined)	
FPSCR	XX (undefined)	
CR	CR1	(if Rc=1)

Note

See the Notes that appear with ***fre***[***s***].

Floating Test for software Divide X-form

ftdiv BF,FRA,FRB



Let e_a be the unbiased exponent of the double-precision floating-point operand in register FRA.

Let e_b be the unbiased exponent of the double-precision floating-point operand in register FRB.

fe_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***[***s***][**.**] of less than or equal to 2^{-14}, then fl_flag is set to 1. Otherwise fl_flag is set to 0.

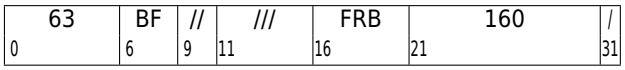
CR field BF is set to the value fl_flag || fg_flag || fe_flag || 0b0.

Special Registers Altered:

Register	*Field(s)*
CR	field BF

Floating Test for software Square Root X-form

ftsqrt BF,FRB



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***[***s***][**.**] 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:

Register	*Field(s)*
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.

4.6.6.2 Floating-Point Multiply-Add Instructions

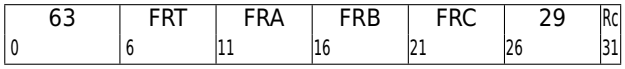
These instructions combine a multiply and an add operation without an intermediate rounding operation. The fraction part of the intermediate product is 106 bits wide (L bit, FRACTION), and all 106 bits take part in the add/-subtract portion of the instruction.

Status bits are set as follows.

- Overflow, Underflow, and Inexact Exception bits, the FR and FI bits, and the FPRF field are set based on the final result of the operation, and not on the result of the multiplication.
- Invalid Operation Exception bits are set as if the multiplication and the addition were performed using two separate instructions (***fmul***[***s***], followed by ***fadd***[***s***] or ***fsub***[***s***]). That is, multiplication of infinity by 0 or of anything by an SNaN, and/or addition of an SNaN, cause the corresponding exception bits to be set.

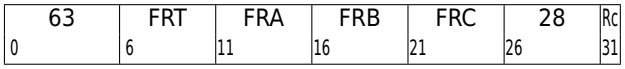
Floating Multiply-Add A-form

fmadd	FRT,FRA,FRC,FRB	(Rc=0)
fmadd.	FRT,FRA,FRC,FRB	(Rc=1)



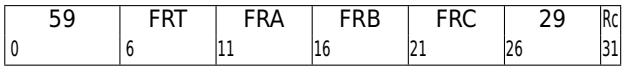
Floating Multiply-Subtract A-form

fmsub	FRT,FRA,FRC,FRB	(Rc=0)
fmsub.	FRT,FRA,FRC,FRB	(Rc=1)



Floating Multiply-Add Single A-form

fmadds	FRT,FRA,FRC,FRB	(Rc=0)
fmadds.	FRT,FRA,FRC,FRB	(Rc=1)



The operation

```
FRT ← [(FRA)×(FRC)] + (FRB)
```

is performed.

The floating-point operand in register FRA is multiplied by the floating-point operand in register FRC. The floating-point operand in register FRB is added to this intermediate result.

If the most significant bit of the resultant significand is not 1, the result is normalized. The result is rounded to the target precision under control of RN and placed into register FRT.

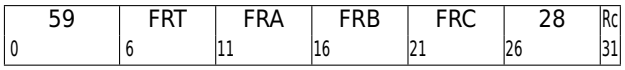
FPRF is set to the class and sign of the result, except for Invalid Operation Exceptions when VE=1.

Special Registers Altered:

Register	*Field(s)*	
FPSCR	FPRF FR FI FX OX UX XX VXSNAN VXISI VXIMZ	
CR	CR1	(if Rc=1)

Floating Multiply-Subtract Single A-form

fmsubs	FRT,FRA,FRC,FRB	(Rc=0)
fmsubs.	FRT,FRA,FRC,FRB	(Rc=1)



The operation

```
FRT ← [(FRA)×(FRC)] - (FRB)
```

is performed.

The floating-point operand in register FRA is multiplied by the floating-point operand in register FRC. The floating-point operand in register FRB is subtracted from this intermediate result.

If the most significant bit of the resultant significand is not 1, the result is normalized. The result is rounded to the target precision under control of RN and placed into register FRT.

FPRF is set to the class and sign of the result, except for Invalid Operation Exceptions when VE=1.

Special Registers Altered:

Register	*Field(s)*	
FPSCR	FPRF FR FI FX OX UX XX VXSNAN VXISI VXIMZ	
CR	CR1	(if Rc=1)

Floating Negative Multiply-Add A-form

fnmadd	FRT,FRA,FRC,FRB	(Rc=0)
fnmadd.	FRT,FRA,FRC,FRB	(Rc=1)

63	FRT	FRA	FRB	FRC	31	Rc
0	6	11	16	21	26	31

Floating Negative Multiply-Add Single A-form

fnmadds	FRT,FRA,FRC,FRB	(Rc=0)
fnmadds.	FRT,FRA,FRC,FRB	(Rc=1)

59	FRT	FRA	FRB	FRC	31	Rc
0	6	11	16	21	26	31

The operation

```
FRT ← – ( [(FRA)×(FRC)] + (FRB) )
```

is performed.

The floating-point operand in register FRA is multiplied by the floating-point operand in register FRC. The floating-point operand in register FRB is added to this intermediate result.

If the most significant bit of the resultant significand is not 1, the result is normalized. The result is rounded to the target precision under control of RN, then negated and placed into register FRT.

This instruction produces the same result as would be obtained by using the *Floating Multiply-Add* instruction and then negating the result, with the following exceptions.

- QNaNs propagate with no effect on their “sign” bit.
- QNaNs that are generated as the result of a disabled Invalid Operation Exception have a “sign” bit of 0.
- SNaNs that are converted to QNaNs as the result of a disabled Invalid Operation Exception retain the “sign” bit of the SNaN.

FPRF is set to the class and sign of the result, except for Invalid Operation Exceptions when VE=1.

Special Registers Altered:

Register	*Field(s)*	
FPSCR	FPRF FR FI FX OX UX XX VXSNAN VXISI VXIMZ	
CR	CR1	(if Rc=1)

Floating Negative Multiply-Subtract A-form

fnmsub	FRT,FRA,FRC,FRB	(Rc=0)
fnmsub.	FRT,FRA,FRC,FRB	(Rc=1)

63	FRT	FRA	FRB	FRC	30	Rc
0	6	11	16	21	26	31

Floating Negative Multiply-Subtract Single A-form

fnmsubs	FRT,FRA,FRC,FRB	(Rc=0)
fnmsubs.	FRT,FRA,FRC,FRB	(Rc=1)

59	FRT	FRA	FRB	FRC	30	Rc
0	6	11	16	21	26	31

The operation

```
FRT ← – ( [(FRA)×(FRC)] – (FRB) )
```

is performed.

The floating-point operand in register FRA is multiplied by the floating-point operand in register FRC. The floating-point operand in register FRB is subtracted from this intermediate result.

If the most significant bit of the resultant significand is not 1, the result is normalized. The result is rounded to the target precision under control of RN, then negated and placed into register FRT.

This instruction produces the same result as would be obtained by using the *Floating Multiply-Subtract* instruction and then negating the result, with the following exceptions.

- QNaNs propagate with no effect on their “sign” bit.
- QNaNs that are generated as the result of a disabled Invalid Operation Exception have a “sign” bit of 0.
- SNaNs that are converted to QNaNs as the result of a disabled Invalid Operation Exception retain the “sign” bit of the SNaN.

FPRF is set to the class and sign of the result, except for Invalid Operation Exceptions when VE=1.

Special Registers Altered:

Register	*Field(s)*	
FPSCR	FPRF FR FI FX OX UX XX VXSNAN VXISI VXIMZ	
CR	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)

63	FRT	///	FRB	12	Rc
0	6	11	16	21	31

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 937.

FPRF is set to the class and sign of the result, except for Invalid Operation Exceptions when VE=1.

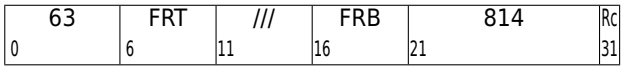
Special Registers Altered:

Register	*Field(s)*	
FPSCR	FPRF FR FI FX OX UX XX VXSNAN	
CR	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

fctid	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 $2^{63}-1$, 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 942.

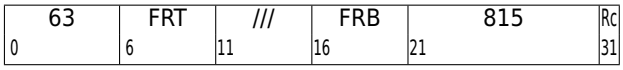
Except for enabled Invalid Operation Exceptions, FPRF is undefined. FR is set if the result is incremented when rounded. FI is set if the result is inexact.

Special Registers Altered:

Register	*Field(s)*	
FPSCR	FR FI FX XX VXSNAN VXCVI	
FPSCR	FPRF (undefined)	
CR	CR1	(if Rc=1)

Floating Convert with truncate Double-Precision To Signed Doubleword format X-form

fctidz	FRT,FRB	(Rc=0)
fctidz.	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 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 942.

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:

Register	*Field(s)*	
FPSCR	FR FI FX XX VXSNAN VXCVI	
FPSCR	FPRF (undefined)	
CR	CR1	(if Rc=1)

Floating Convert with round Double-Precision To Unsigned Doubleword format X-form

fctidu	FRT,FRB	(Rc=0)
fctidu.	FRT,FRB	(Rc=1)

63	FRT	///	FRB	942	Rc
0	6	11	16	21	31

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 942.

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:

Register	*Field(s)*	
FPSCR	FR FI FX XX VXSNAN VXCVI	
FPSCR	FPRF (undefined)	
CR	CR1	(if Rc=1)

Floating Convert with truncate Double-Precision To Unsigned Doubleword format X-form

fctiduz	FRT,FRB	(Rc=0)
fctiduz.	FRT,FRB	(Rc=1)

63	FRT	///	FRB	943	Rc
0	6	11	16	21	31

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 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 942.

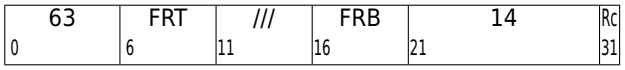
Except for enabled Invalid Operation Exceptions, FPRF is undefined. FR is set if the result is incremented when rounded. FI is set if the result is inexact.

Special Registers Altered:

Register	*Field(s)*	
FPSCR	FR FI FX XX VXSNAN VXCVI	
FPSCR	FPRF (undefined)	
CR	CR1	(if Rc=1)

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 942.

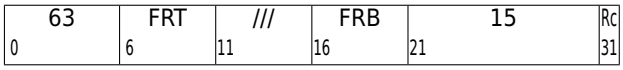
Except for enabled Invalid Operation Exceptions, FPRF is undefined. FR is set if the result is incremented when rounded. FI is set if the result is inexact.

Special Registers Altered:

Register	*Field(s)*	
FPSCR	FR FI FX XX VXSNAN VXCVI	
FPSCR	FPRF (undefined)	
CR	CR1	(if Rc=1)

Floating Convert with truncate Double-Precision To Signed Word fomat 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 $2^{31}-1$, 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 942.

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:

Register	*Field(s)*	
FPSCR	FR FI FX XX VXSNAN VXCVI	
FPSCR	FPRF (undefined)	
CR	CR1	(if Rc=1)

Floating Convert with round Double-Precision To Unsigned Word format X-form

fctiwu	FRT,FRB	(Rc=0)
fctiwu.	FRT,FRB	(Rc=1)

63	FRT	///	FRB	142	Rc
0	6	11	16	21	31

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 $2^{32}-1$, then the result is 0xFFFF_FFFF and VXCVI is set to 1.

Otherwise, if the rounded value is less than 0, then the result is 0x0000_0000 and VXCVI is set to 1.

Otherwise, the result is the rounded value converted to 32-bit unsigned-integer format, and XX is set to 1 if the result is inexact.

If an enabled Invalid Operation Exception does not occur, then the result is placed into $FRT_{32:63}$ and $FRT_{0:31}$ is undefined,

The conversion is described fully in Section A.2, “Floating-Point Convert to Integer Model” on page 942.

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:

Register	*Field(s)*	
FPSCR	FR FI FX XX VXSNAN VXCVI	
FPSCR	FPRF (undefined)	
CR	CR1	(if Rc=1)

Floating Convert with truncate Double-Precision To Unsigned Word format X-form

fctiwuz	FRT,FRB	(Rc=0)
fctiwuz.	FRT,FRB	(Rc=1)

63	FRT	///	FRB	143	Rc
0	6	11	16	21	31

Let src be the double-precision floating-point value in FRB.

If src is a NaN, then the result is 0x0000_0000, VXCVI is set to 1, and, if src is an SNaN, VXSNAN is set to 1.

Otherwise, src is rounded to a floating-point integer using the rounding mode Round toward Zero.

If the rounded value is greater than $2^{32}-1$, then the result is 0xFFFF_FFFF and VXCVI is set to 1.

Otherwise, if the rounded value is less than 0.0, then the result is 0x0000_0000 and VXCVI is set to 1.

Otherwise, the result is the rounded value converted to 32-bit unsigned-integer format, and XX is set to 1 if the result is inexact.

If an enabled Invalid Operation Exception does not occur, then the result is placed into $FRT_{32:63}$ and $FRT_{0:31}$ is undefined,

The conversion is described fully in Section A.2, “Floating-Point Convert to Integer Model” on page 942.

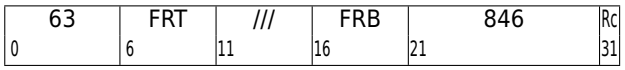
Except for enabled Invalid Operation Exceptions, FPRF is undefined. FR is set if the result is incremented when rounded. FI is set if the result is inexact.

Special Registers Altered:

Register	*Field(s)*	
FPSCR	FR FI FX XX VXSNAN VXCVI	
FPSCR	FPRF (undefined)	
CR	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)



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:

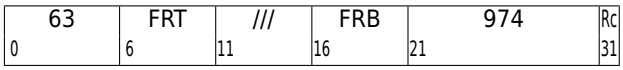
Register	*Field(s)*	
FPSCR	FPRF FR FI FX XX	
CR	CR1	(if Rc=1)

Programming Note

Converting a signed integer word to double-precision floating-point can be accomplished by loading the word from storage using *Load Float Word Algebraic Indexed* and then using ***fcfid***.

Floating Convert with round Unsigned Doubleword to Double-Precision format X-form

fcfidu	FRT,FRB	(Rc=0)
fcfidu.	FRT,FRB	(Rc=1)



The 64-bit unsigned fixed-point operand in register FRB is converted to an infinitely precise floating-point integer. The result of the conversion is rounded to double-precision, using the rounding mode specified by RN, and placed into register FRT.

The conversion is described fully in Section A.3, “Floating-Point Convert from Integer Model”.

FPRF is set to the class and sign of the result. FR is set if the result is incremented when rounded. FI is set if the result is inexact.

Special Registers Altered:

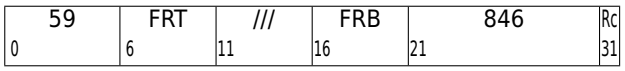
Register	*Field(s)*	
FPSCR	FPRF FR FI FX XX	
CR	CR1	(if Rc=1)

Programming Note

Converting an unsigned integer word to double-precision floating-point can be accomplished by loading the word from storage using *Load Float Word and Zero Indexed* and then using ***fcfidu***.

Floating Convert with round Signed Doubleword to Single-Precision format X-form

fcfids	FRT,FRB	(Rc=0)
fcfids.	FRT,FRB	(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 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:

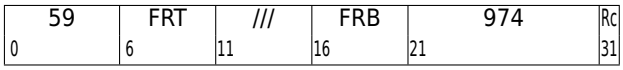
Register	*Field(s)*	
FPSCR	FPRF FR FI FX XX	
CR	CR1	(if Rc=1)

Programming Note

Converting a signed integer word to single-precision floating-point can be accomplished by loading the word from storage using *Load Float Word Algebraic Indexed* and then using ***fcfids***.

Floating Convert with round Unsigned Doubleword to Single-Precision format X-form

fcfidus	FRT,FRB	(Rc=0)
fcfidus.	FRT,FRB	(Rc=1)



The 64-bit unsigned fixed-point operand in register FRB is converted to an infinitely precise floating-point integer. The result of the conversion is rounded to single-precision, using the rounding mode specified by RN, and placed into register FRT.

The conversion is described fully in Section A.3, "Floating-Point Convert from Integer Model".

FPRF is set to the class and sign of the result. FR is set if the result is incremented when rounded. FI is set if the result is inexact.

Special Registers Altered:

Register	*Field(s)*	
FPSCR	FPRF FR FI FX XX	
CR	CR1	(if Rc=1)

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 948.

Programming Note

These instructions set FR and FI to 0b00 regardless of whether the result is inexact or rounded because there is a desire to preserve the value of XX. Furthermore, it is believed that most programs do not need to know whether these rounding operations produce inexact or rounded results. If it is necessary to determine whether the result is inexact or rounded, software must compare the result with the original source operand.

Architecture Note

Single-precision versions of these instructions are not provided because they would be superfluous: if the contents of FRB are representable in single format, then so are the contents of FRT.

Architecture Note

These instructions were originally placed in column 01100 with ***frsp***. Because all of that column except ***frsp*** are part of Book E's "allocated" opcode space, the instructions were moved to column 01000.

Floating Round to Integer Nearest X-form

frin	FRT,FRB	(Rc=0)
frin.	FRT,FRB	(Rc=1)

63	FRT	///	FRB	392	Rc
0	6	11	16	21	31

The floating-point operand in register FRB is rounded to an integral value as follows, with the result placed into register FRT. If the sign of the operand is positive, (FRB) + 0.5 is truncated to an integral value, otherwise (FRB) – 0.5 is truncated to an integral value.

FPRF is set to the class and sign of the result, except for Invalid Operation Exceptions when VE=1.

Special Registers Altered:

Register	*Field(s)*	
FPSCR	FPRF FX VXSNAN	
FPSCR	FR (set to 0)	
FPSCR	FI (set to 0)	
CR	CR1	(if Rc=1)

Floating Round to Integer Toward Zero X-form

friz	FRT,FRB	(Rc=0)
friz.	FRT,FRB	(Rc=1)

63	FRT	///	FRB	424	Rc
0	6	11	16	21	31

The floating-point operand in register FRB is rounded to an integral value using the rounding mode round toward zero, and the result is placed into register FRT.

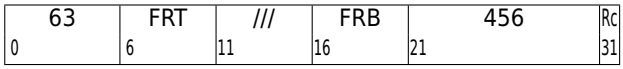
FPRF is set to the class and sign of the result, except for Invalid Operation Exceptions when VE = 1.

Special Registers Altered:

Register	*Field(s)*	
FPSCR	FPRF FX VXSNAN	
FPSCR	FR (set to 0)	
FPSCR	FI (set to 0)	
CR	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.

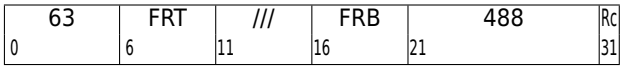
FPRF is set to the class and sign of the result, except for Invalid Operation Exceptions when VE=1.

Special Registers Altered:

Register	*Field(s)*	
FPSCR	FPRF FX VXSNAN	
FPSCR	FR (set to 0)	
FPSCR	FI (set to 0)	
CR	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.

FPRF is set to the class and sign of the result, except for Invalid Operation Exceptions when VE=1.

Special Registers Altered:

Register	*Field(s)*	
FPSCR	FPRF FX VXSNAN	
FPSCR	FR (set to 0)	
FPSCR	FI (set to 0)	
CR	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.

Bit	Name	Description
0	FL	(FRA) $<$ (FRB)
1	FG	(FRA) $>$ (FRB)
2	FE	(FRA) = (FRB)
3	FU	(FRA) ? (FRB) (unordered)

Floating Compare Unordered X-form

fcmpu BF,FRA,FRB

63	BF	//	FRA	FRB	0	/
0	6	9	11	16	21	31

```
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:

Register	*Field(s)*
FPSCR	FPCC FX VXSNAN
CR	field BF

Floating Compare Ordered X-form

fcmpo BF,FRA,FRB

63	BF	//	FRA	FRB	32	/
0	6	9	11	16	21	31

```
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:

Register	*Field(s)*
FPSCR	FPCC FX VXSNAN
	VXVC
CR	field BF

4.6.9 Floating-Point Select Instruction

Floating Select A-form

fsel	FRT,FRA,FRC,FRB	(Rc=0)
fsel.	FRT,FRA,FRC,FRB	(Rc=1)

63	FRT	FRA	FRB	FRC	23	Rc
0	6	11	16	21	26	31

```
if (FRA) ≥ 0.0 then FRT ← (FRC)
else FRT ← (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:

Register	*Field(s)*	
CR	CR1	(if Rc=1)

Architecture Note

The Select instruction is similar to a Move instruction, and therefore does not alter the FPSCR.

Programming Note

Warning: Care must be taken in using ***fsel*** if IEEE compatibility is required, or if the values being tested can be NaNs or infinities.

Programming Note

This Note gives examples of how the *Floating Select* instruction can be used to implement certain simple forms of if-then-else constructions, without branching.

The examples show program fragments in an imaginary, C-like, high-level programming language, and the corresponding program fragment using ***fsel*** and other Power ISA instructions. In the examples, `a`, `b`, `x`, `y`, and `z` are floating-point variables, which are assumed to be in FPRs `fa`, `fb`, `fx`, `fy`, and `fz`. FPR `fs` is assumed to be available for scratch space.

Warning: Care must be taken in using ***fsel*** if IEEE compatibility is required, or if the values being tested can be NaNs or infinities; see the Notes below.

Comparison to Zero

High-level language:	Power ISA:	Notes

`if a ≥ 0.0 then`
`  x ← y`
`else`
`  x ← z`
	`fsel  fx,fa,fy,fz`	(1)

`if a > 0.0 then`
`  x ← y`
`else`
`  x ← z`
	`fneg  fs,fa`
	`fsel  fx,fs,fz,fy`
		(1,2)

`if a = 0.0 then`
`  x ← y`
`else`
`  x ← z`
	`fsel  fx,fa,fy,fz`
	`fneg  fs,fa`
	`fsel  fx,fs,fx,fz`
		(1)

Minimum and Maximum

High-level language:	Power ISA:	Notes
`x ← min(a,b)`	`fsub  fs,fa,fb` `fsel  fx,fs,fb,fa`	(3,4,5)
`x ← max(a,b)`	`fsub  fs,fa,fb` `fsel  fx,fs,fa,fb`	(3,4,5)

Simple if-then-else Constructions

High-level language:	Power ISA:	Notes
`if a ≥ b then x ← y` `else x ← z`	`fsub  fs,fa,fb` `fsel  fx,fs,fy,fz`	(4,5)
`if a > b then x ← y` `else x ← z`	`fsub  fs,fb,fa` `fsel  fx,fs,fz,fy`	(3,4,5)

`if a = b then x ← y`
`else x ← z`
	`fsub  fs,fa,fb`
	`fsel  fx,fs,fy,fz`
	`fneg  fs,fs`
	`fsel  fx,fs,fx,fz`
		(4,5)

Notes:

The following Notes apply to the preceding examples and to the corresponding cases using the other three arithmetic relations ($<$, $\leq$, and $\neq$). They should also be considered when any other use of ***fsel*** is contemplated.

In these Notes, the "optimized program" is the Power ISA program shown, and the "unoptimized program" (not shown) is the corresponding Power ISA program that uses ***fcmpu*** and *Branch Conditional* instructions instead of ***fsel***.

1. The unoptimized program affects the VXSNAN bit of the FPSCR, and therefore may cause the system error handler to be invoked if the corresponding exception is enabled, while the optimized program does not affect this bit. This property of the optimized program is incompatible with the IEEE standard.
2. The optimized program gives the incorrect result if a is a NaN.
3. The optimized program gives the incorrect result if a and/or b is a NaN (except that it may give the correct result in some cases for the minimum and maximum functions, depending on how those functions are defined to operate on NaNs).
4. The optimized program gives the incorrect result if a and b are infinities of the same sign. (Here it is assumed that Invalid Operation Exceptions are disabled, in which case the result of the subtraction is a NaN. The analysis is more complicated if Invalid Operation Exceptions are enabled, because in that case the target register of the subtraction is unchanged.)
5. The optimized program affects the OX, UX, XX, and VXISI bits of the FPSCR, and therefore may cause the system error handler to be invoked if the corresponding exceptions are enabled, while the unoptimized program does not affect these bits. This property of the optimized program is incompatible with the IEEE standard.

4.6.10 Floating-Point Status and Control Register Instructions

Except as described below for ***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.

While not satisfying all of the conditions described above, ***mffsce***, ***mffscdrn***[***i***], ***mffscrn***[***i***], and ***mffsl*** still obey the sequential execution model. Any FPSCR status bits read by ***mffsce*** or ***mffsl*** will reflect updates due to all preceding floating-point instructions. That is, all floating-point instructions following an ***mffsce***, ***mffscdrn***[***i***], or ***mffscrn***[***i***] will execute based on any updates applied to any control bits in the FPSCR by the ***mffsce***, ***mffscdrn***[***i***], or ***mffscrn***[***i***].

(Floating-point *Storage Access* instructions are not affected.)

The instruction descriptions in this section refer to "FPSCR fields," where FPSCR field k is FPSCR bits $4\times k:4\times k+3$.

Move From FPSCR X-form

mffs	FRT	(Rc=0)
mffs.	FRT	(Rc=1)

63	FRT	0	///	583	Rc
0	6	11	16	21	31

The contents of the FPSCR are placed into register FRT.

If Rc=1, CR field 1 is set to the value FX||FEX||VX||OX.

Special Registers Altered:

Register	*Field(s)*	
CR	CR1	(if Rc=1)

Move From FPSCR & Clear Enables X-form

mffsce	FRT

63	FRT	1	///	583	/
0	6	11	16	21	31

The contents of the FPSCR are placed into register FRT.

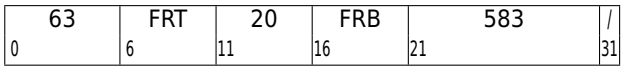
The contents of bits 56:60 (VE, OE, UE, ZE, XE) of the FPSCR are set to 0.

Special Registers Altered:

Register	*Field(s)*
FPSCR	VE (set to 0)
FPSCR	OE (set to 0)
FPSCR	UE (set to 0)
FPSCR	ZE (set to 0)
FPSCR	XE (set to 0)

Move From FPSCR Control & Set DRN X-form

mffscdrn FRT,FRB



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.

new_DRN is placed into bits 62:64 of the FPSCR (DRN).

Special Registers Altered:

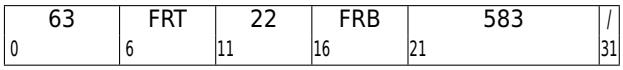
Register	*Field(s)*
FPSCR	DRN

Programming Note

mffscdrn permits software to simultaneously read control bits in the FPSCR and set the DRN field without the higher latency typically associated with accessing the status bits.

Move From FPSCR Control & Set RN X-form

mffscrn FRT,FRB



Let *new_RN* be the contents of bits 62:63 of register FRB.

The contents of the control bits in the FPSCR, that is, bits 29:31 (DRN) and bits 56:63 (VE, OE, UE, ZE, XE, NI, RN), are placed into the corresponding bits in register FRT. All other bits in register FRT are set to 0.

new_RN is placed into bits 62:63 of the FPSCR (RN).

Special Registers Altered:

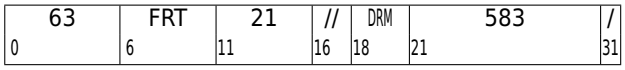
Register	*Field(s)*
FPSCR	RN

Programming Note

mffscrn permits software to simultaneously read control bits in the FPSCR and set the RN field without the higher latency typically associated with accessing the status bits.

Move From FPSCR Control & Set DRN Immediate X-form

mffscdrni FRT,DRM



The contents of the control bits in the FPSCR, that is, bits 29:31 (DRN) and bits 56:63 (VE, OE, UE, ZE, XE, NI, RN), are placed into the corresponding bits in register FRT. All other bits in register FRT are set to 0.

The contents of bits 29:31 of the FPSCR (DRN) are set to the value of DRM.

Special Registers Altered:

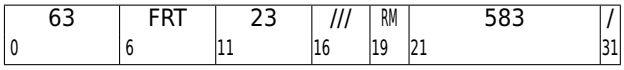
Register	*Field(s)*
FPSCR	DRN

Programming Note

mffscdrni permits software to simultaneously read control bits in the FPSCR and set the DRN field without the higher latency typically associated with accessing the status bits.

Move From FPSCR Control & Set RN Immediate X-form

mffscrni FRT,RM



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.

Special Registers Altered:

Register	*Field(s)*
FPSCR	RN

Programming Note

mffscrni permits software to simultaneously read control bits in the FPSCR and set the RN field without the higher latency typically associated with accessing the status bits.

Move From FPSCR Lightweight X-form

mffsl FRT

63	FRT	24	///	583	/
0	6	11	16	21	31

The contents of the control bits in the FPSCR, that is, bits 29:31 (DRN) and bits 56:63 (VE, OE, UE, ZE, XE, NI, RN), and the non-sticky status bits in the FPSCR, that is, bits 45:51 (FR, FI, C, FL, FG, FE, FU), are placed into the corresponding bits in register FRT. All other bits in register FRT are set to 0.

Special Registers Altered:

None

Programming Note

mffsl permits software to read the control and non-sticky status bits in the FPSCR without the higher latency typically associated with accessing the sticky status bits.

Move to Condition Register from FPSCR X-form

mcrfs BF,BFA

63	BF	//	BFA	//	///	64	/
0	6	9	11	14	16	21	31

The contents of $FPSCR_{32:63}$ 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.

Special Registers Altered:

Register	*Field(s)*	
CR	field BF	
FPSCR	FX OX	(if BFA=0)
FPSCR	UX ZX XX VXSNAN	(if BFA=1)
FPSCR	VXISI VXIDI VXZDZ VXIMZ	(if BFA=2)
FPSCR	VXVC	(if BFA=3)
FPSCR	VXSOFT VXSQRT VXCVI	(if BFA=5)

Move To FPSCR Field Immediate X-form

mtfsfi BF,U,W (Rc=0)
mtfsfi. BF,U,W (Rc=1)

63	BF	//	///	W	U	/	134	Rc
0	6	9	11	15	16	20	21	31

The value of the U field is placed into FPSCR field BF+8*(1-W).

FX is altered only if BF=0 and W=0.

Special Registers Altered:

Register	*Field(s)*	
FPSCR	field BF + 8*(1-W)	
CR	CR1	(if Rc=1)

Extended Mnemonics:

Extended Mnemonics for *Move to FPSCR Field Immediate*:

Extended mnemonic:	Equivalent to:
mtfsfi BF,U	mtfsfi BF,U,0

Programming Note

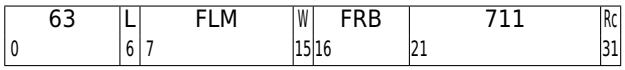
mtfsfi serves as both a basic and an extended mnemonic. The Assembler will recognize a ***mtfsfi*** mnemonic with three operands as the basic form, and a ***mtfsfi*** mnemonic with two operands as the extended form. In the extended form the W operand is omitted and assumed to be 0.

Programming Note

When $FPSCR_{32:35}$ is specified, bits 32 (FX) and 35 (OX) are set to the values of U_0 and U_3 (i.e., even if this instruction causes OX to change from 0 to 1, FX is set from U_0 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 134, and not from $U_{1:2}$.

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.

FX is not altered implicitly by this instruction.

Special Registers Altered:

Register	*Field(s)*	
FPSCR	fields selected by mask, L, and W	
CR	CR1	(if Rc=1)

Extended Mnemonics:

Extended Mnemonics for *Move to FPSCR Fields*:

Extended mnemonic:	Equivalent to:
mtfsf FLM,FRB	mtfsf FLM,FRB,0,0

Programming Note

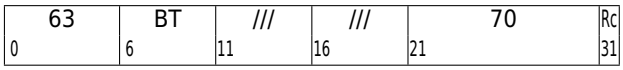
mtfsf serves as both a basic and an extended mnemonic. The Assembler will recognize a ***mtfsf*** mnemonic with four operands as the basic form, and a ***mtfsf*** mnemonic with two operands as the extended form. In the extended form the W and L operands are omitted and both are assumed to be 0.

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 134, and not from $(FRB)_{33:34}$.

Move To FPSCR Bit 0 X-form

mtfsb0	BT	(Rc=0)
mtfsb0.	BT	(Rc=1)



Bit BT+32 of the FPSCR is set to 0.

Special Registers Altered:

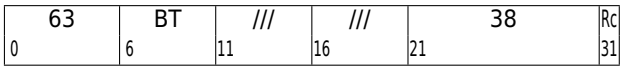
Register	*Field(s)*	
FPSCR	bit BT+32	
CR	CR1	(if Rc=1)

Programming Note

Bits 33 and 34 (FEX and VX) cannot be explicitly reset.

Move To FPSCR Bit 1 X-form

mtfsb1	BT	(Rc=0)
mtfsb1.	BT	(Rc=1)



Bit BT+32 of the FPSCR is set to 1.

Special Registers Altered:

Register	*Field(s)*	
FPSCR	bit BT+32	
FPSCR	FX	
CR	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 faciltiy, and the vector facility.

The DFP facility supports three DFP data formats: DFP Short (single precision), DFP Long (double precision), and DFP Extended (quad precision). Most operations are performed on DFP Long or DFP Extended format directly. Support for DFP Short is limited to conversion to and from DFP Long. Some DFP instructions operate on other data types, including signed or unsigned binary fixed-point data, and signed or unsigned decimal data.

DFP instructions are provided to perform arithmetic, compare, test, quantum-adjustment, conversion, and format operations on operands held in FPRs or FPR pairs.

- Arithmetic instructions
  These instructions perform addition, subtraction, multiplication, and division operations.
- Compare instructions
  These instructions perform a comparison operation on the numerical value of two DFP operands.
- Test instructions
  These instructions test the data class, the data group, the exponent, or the number of significant digits of a DFP operand.
- Quantum-adjustment instructions
  These instructions convert a DFP number to a result in the form that has the designated exponent, which may be explicitly or implicitly specified.
- Conversion instructions
  These instructions perform conversion between different data formats or data types.
- Format instructions
  These instructions facilitate composing or decomposing a DFP operand.

These instructions are described in Section 5.6 “DFP Instruction Descriptions” on page 204.

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)
  - $\infty - \infty$ (VXISI)
  - $\infty \div \infty$ (VXIDI)
  - $0 \div 0$ (VXZDZ)
  - $\infty \times 0$ (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 196 provides a detailed discussion of DFP exceptions, including the effects of the enable bits.

5.2 DFP Register Handling

The following sections describe first how the floating-point registers are utilized by the DFP facility. The subsequent section covers the DFP usage of CR and FPSCR.

5.2.1 DFP Usage of Floating-Point Registers

The DFP facility shares the same 32 64-bit FPRs with the BFP facility. Like the FP instructions, DFP instructions also use 5-bit fields for designating the FPRs to hold the source or target operands.

When data in DFP Short format is held in a FPR, it occupies the rightmost 32 bits of the FPR. The *Load Floating-Point as Integer Word Algebraic* instruction is provided to load the rightmost 32 bits of a FPR with a single-word data from storage. The *Store Floating-Point as Integer Word* instruction is available to store the rightmost 32 bits of a FPR to a storage location.

Data in DFP Long format, 64-bit binary fixed-point values, or 64-bit BCD values is held in a FPR using all 64 bits. Data of 64 bits may be loaded from storage via any of the *Load Floating-Point Double* instructions and stored via any of the *Store Floating-Point Double* instructions.

Data in DFP Extended format or 128-bit BCD values is held in an even-odd FPR pair using all 128 bits. Data of 128 bits must be loaded into the desired even-odd pair of floating-point registers using an appropriate sequence of the *Load Floating-Point Double* instructions and stored using an appropriate sequence of the *Store Floating-Point Double* instructions.

Data used as a source operand by any *Decimal Floating-Point* instruction that was produced, either directly or indirectly, by a *Load Floating-Point Single* instruction, a *Floating Round to Single-Precision* instruction, or a binary floating-point single-precision arithmetic instruction is boundedly undefined.

When an even-odd FPR pair is used to hold a 128-bit operand, the even-numbered FPR is used to hold the leftmost doubleword of the operand and the next higher-numbered FPR is used to hold the rightmost doubleword. A DFP instruction designating an odd-numbered FPR for a 128-bit operand is an invalid instruction form.

Programming Note

The *Floating-Point Move* instructions can be used to move operands between FPRs.

The bit definitions for the FPSCR are as follows.

Bit(s)	Definition
0:28	Reserved
29:31	**DFP Rounding Control** (DRN) See Section 5.5.2, “Rounding Mode Specification” on page 193. 000 Round to Nearest, Ties to Even 001 Round toward Zero 010 Round toward +Infinity 011 Round toward −Infinity 100 Round to Nearest, Ties away from 0 101 Round to Nearest, Ties toward 0 110 Round to away from Zero 111 Round to Prepare for Shorter Precision

Programming Note

$FPSCR_{28}$ is reserved for extension of the DRN field, therefore DRN may be set using the ***mtfsfi*** instruction to set the rounding mode.

Architecture Note

$FPSCR_{28}$ should be regarded as reserved for extension of the DRN field.

32	**Floating-Point Exception Summary** (FX) Every floating-point instruction, except ***mtfsfi*** and ***mtfsf***, implicitly sets FX to 1 if that instruction causes any of the floating-point exception bits in the FPSCR to change from 0 to 1. ***mcrfs***, ***mtfsfi***, ***mtfsf***, ***mtfsb0***, and ***mtfsb1*** can alter FX explicitly.
33	**Floating-Point Enabled Exception Summary** (FEX) This bit is the OR of all the floating-point exception bits masked by their respective enable bits. ***mcrfs***, ***mtfsfi***, ***mtfsf***, ***mtfsb0***, and ***mtfsb1*** cannot alter FEX explicitly.
34	**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 VX explicitly.
35	**Floating-Point Overflow Exception** (OX) See Section 5.5.10.3, “Overflow Exception” on page 199.
36	**Floating-Point Underflow Exception** (UX) See Section 5.5.10.4, “Underflow Exception” on page 200.
37	**Floating-Point Zero Divide Exception** (ZX) See Section 5.5.10.2, “Zero Divide Exception” on page 199.

38 **Floating-Point Inexact Exception** (XX)

See Section 5.5.10.5, "Inexact Exception" on page 200.

XX is a sticky version of FI (see below). Thus the following rules completely describe how XX is set by a given instruction.

- If the instruction affects FI, the new value of XX is obtained by ORing the old value of XX with the new value of FI.
- If the instruction does not affect FI, the value of XX is unchanged.

39 **Floating-Point Invalid Operation Exception (SNaN)** (VXSNAN)

See Section 5.5.10.1, "Invalid Operation Exception" on page 198.

40 **Floating-Point Invalid Operation Exception (Infinity − Infinity)** (VXISI)

See Section 5.5.10.1.

41 **Floating-Point Invalid Operation Exception (Infinity ÷ Infinity)** (VXIDI)

See Section 5.5.10.1.

42 **Floating-Point Invalid Operation Exception (Zero ÷ Zero)** (VXZDZ)

See Section 5.5.10.1.

43 **Floating-Point Invalid Operation Exception (Infinity × Zero)** (VXIMZ)

See Section 5.5.10.1.

44 **Floating-Point Invalid Operation Exception (Invalid Compare)** (VXVC)

See Section 5.5.10.1.

45 **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 192. This bit is not sticky.

46 **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.

See the definition of XX, above, regarding the relationship between FI and XX.

47:51 **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.

47 **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 5.1 on page 189.

48:51 **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 5.1 on page 189. Note that in this case the high-order three bits of the FPCC retain their relational significance indicating that the value is less than, greater than, or equal to zero.

48 **Floating-Point Less Than or Negative** (FL or <)

49 **Floating-Point Greater Than or Positive** (FG or >)

50 **Floating-Point Equal or Zero** (FE or =)

51 **Floating-Point Unordered or NaN** (FU or ?)

52 Reserved

53 **Floating-Point Invalid Operation Exception (Software Request)** (VXSOFT)

This bit can be altered only by ***mcrfs***, ***mtfsfi***, ***mtfsf***, ***mtfsb0***, or ***mtfsb1***. See Section 5.5.10.1, "Invalid Operation Exception" on page 198.

54 Reserved

Neither used nor changed by DFP.

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.

55 **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 199.

58 **Floating-Point Underflow Exception Enable** (UE)

See Section 5.5.10.4, "Underflow Exception" on page 200.

59 **Floating-Point Zero Divide Exception Enable** (ZE)

See Section 5.5.10.2, "Zero Divide Exception" on page 199.

60 **Floating-Point Inexact Exception Enable** (XE)

See Section 5.5.10.5, "Inexact Exception" on page 200

61 Reserved

Not used by DFP.

62:63 **Binary Floating-Point Rounding Control** (RN)

See Section 5.5.1, "Rounding" on page 192.

00 Round to Nearest
01 Round toward Zero
10 Round toward +Infinity
11 Round toward −Infinity

Result Flags					Result Value Class
C	<	>	=	?	
0	0	0	0	1	Signaling NaN (DFP only)
1	0	0	0	1	Quiet NaN
0	1	0	0	1	− Infinity
0	1	0	0	0	− Normal Number
1	1	0	0	0	− Subnormal Number
1	0	0	1	0	− Zero
0	0	0	1	0	+ Zero
1	0	1	0	0	+ Subnormal Number
0	0	1	0	0	+ Normal Number
0	0	1	0	1	+ Infinity

Figure 5.1: Floating-Point Result Flags

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 5.2

D	D	D	D	. . .	D	D	D	D

Figure 5.2: 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 5.3.

D	D	D	D	. . .	D	D	D	S

Figure 5.3: 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-operation exception occurs. A summary of digit and sign codes are provided in Figure 5.4.

Binary Code	Recognized As	
	Digit	Sign
0000	0	Invalid
0001	1	Invalid
0010	2	Invalid
0011	3	Invalid
0100	4	Invalid
0101	5	Invalid
0110	6	Invalid
0111	7	Invalid
1000	8	Invalid
1001	9	Invalid
1010	Invalid	Plus
1011	Invalid	Minus
1100	Invalid	Plus (preferred; option 1)
1101	Invalid	Minus (preferred)
1110	Invalid	Plus
1111	Invalid	Plus (preferred; option 2)

Figure 5.4: 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 $(-1)^{\texttt{sign}} \times \texttt{significand} \times 10^{\texttt{exponent}}$ and the unit value of this number is $(1 \times 10^{\texttt{exponent}})$, 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 nonzero 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:

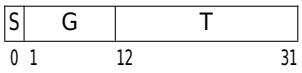


Figure 5.5: DFP Short format

Figure 5.6: DFP Long format

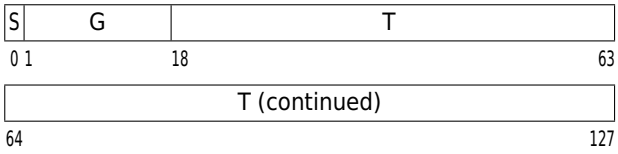


Figure 5.7: DFP Extended format

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:

$G_{0:4}$	Description
11111	NaN
11110	Infinity
All others	Finite Number (see Figure 5.9)

Figure 5.8: Encoding of the G field for Special Symbols

LMD	Leftmost 2-bits of biased exponent		
	00	01	10
0	00000	01000	10000
1	00001	01001	10001
2	00010	01010	10010
3	00011	01011	10011
4	00100	01100	10100
5	00101	01101	10101
6	00110	01110	10110
7	00111	01111	10111
8	11000	11010	11100
9	11001	11011	11101

Figure 5.9: 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.

	Format		
	DFP Short	**DFP Long**	**DFP Extended**
Widths (bits):			
Format	32	64	128
Sign (s)	1	1	1
Combination (g)	11	13	17
Trailing Significand (t)	20	50	110
Exponent:			
Maximum biased	191	767	12,287
Maximum (X_{max})	90	369	6111
Minimum (X_{min})	-101	-398	-6176
Bias	101	398	6176
Precision (p) (digits)	7	16	34
Magnitude:			
Maximum normal number (N_{max})	$(10^7 - 1) \times 10^{90}$	$(10^{16} - 1) \times 10^{369}$	$(10^{34} - 1) \times 10^{6111}$
Minimum normal number (N_{min})	1×10^{-95}	1×10^{-383}	1×10^{-6143}
Minimum subnormal number (D_{min})	1×10^{-101}	1×10^{-398}	1×10^{-6176}

Figure 5.10: Summary of DFP Formats

5.4.1.3 Preferred DPD Encoding

Execution of DFP instructions decodes source operands from DFP data formats to an internal format for processing, and encodes the operation result before the final result is returned as the target operand.

As part of the decoding process, declets in the trailing significand field of source operands are decoded to their corresponding BCD digit codes using the DPD-to-BCD decoding algorithm. As part of the encoding process, BCD digit codes to be stored into the trailing significand field of the target operand are encoded into declets using the BCD-to-DPD encoding algorithm. Both the decoding and encoding algorithms are defined in Appendix B.

As explained in Appendix B, there are eight 3-digit decimal values that have redundant DPD codes and one preferred DPD code. All redundant DPD codes are recognized in source operands for the associated 3-digit decimal number. DFP operations will always generate the preferred DPD codes for the trailing significand field of the target operand.

5.4.2 Classes of DFP Data

There are six classes of DFP data, which include numerical and nonnumeric entities. The numerical entities include zero, subnormal number, normal number, and infinity data classes. The nonnumeric entities include quiet and signaling NaNs data classes. The value of a DFP finite number, including zero, subnormal number, and normal number, is a quantization of the real number based on the data format. The *Test Data Class* instruction may be used to determine the class of a DFP operand. In general, an operation that returns a DFP result sets the FPRF field to indicate the data class of the result.

The following tables show the value ranges for finite-number data classes, and the codes for NaNs and infinities.

Data Class	**Sign**	**Magnitude**
Zero	$\pm$	0*
Subnormal	$\pm$	$D_{min} \leq \lvert X \rvert < N_{min}$
Normal	$\pm$	$N_{min} \leq \lvert Y \rvert \leq N_{max}$
* The significand is zero and the exponent is any representable value		

Figure 5.11: Value Ranges for Finite Number Data Classes

Data Class	**S**	**G**	**T**
+Infinity	0	11110xxx ... xxx	xxx ... xxx
−Infinity	1	11110xxx ... xxx	xxx ... xxx
Quiet NaN	x	111110xx ... xxx	xxx ... xxx
Signaling NaN	x	111111xx ... xxx	xxx ... xxx
x Don't care			

Figure 5.12: 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_{min} and greater than zero in magnitude.

Normal Numbers
Normal numbers are nonzero finite numbers whose magnitude is between N_{min} and N_{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 proagation, 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 is 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 193.)

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 5.13 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.

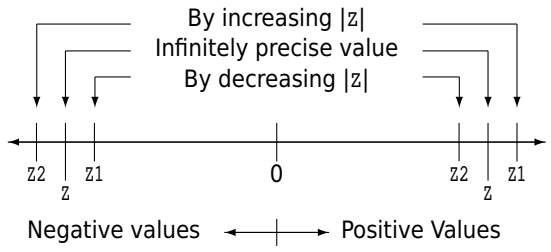


Figure 5.13: Rounding

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_{max} + 0.5Q(N_{max})) is rounded to infinity with no change in sign; where Q(N_{max}) is the quantum of N_{max}.

Round toward 0
Choose the smaller in magnitude (Z1 or Z2).

Round toward $+\infty$
Choose Z1.

Round toward $-\infty$
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_{max} + 0.5Q(N_{max})) is rounded to infinity with no change in sign; where Q(N_{max}) is the quantum of N_{max}.

Round to Nearest, Ties toward 0
Choose the value that is closer to Z (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_{max} + 0.5Q(N_{max})) is rounded to infinity with no change in sign; where Q(N_{max}) is the quantum of N_{max}.

Round away from 0
Choose the larger in magnitude (Z1 or Z2).

Round to prepare for shorter precision
Choose the smaller in magnitude (Z1 or Z2). 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.

DRN	Rounding Mode
000	Round to Nearest, Ties to Even
001	Round toward 0
010	Round toward +Infinity
011	Round toward −Infinity
100	Round to Nearest, Ties away from 0
101	Round to Nearest, Ties toward 0
110	Round away from 0
111	Round to Prepare for Shorter Precision

Figure 5.14: 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.

Primary RMC	Rounding Mode
00	Round to nearest, ties to even
01	Round toward 0
10	Round to nearest, ties away from 0
11	Round according to DRN

Figure 5.15: Primary Encoding of Rounding-Mode Control

Secondary RMC	Rounding Mode
00	Round to $+\infty$
01	Round to $-\infty$
10	Round away from 0
11	Round to nearest, ties toward 0

Figure 5.16: Secondary Encoding of Rounding-Mode Control

Engineering Note

Bits 0 and 1 of the primary and secondary RMC encodings are the same as bits 0 and 2, respectively, of the corresponding DRN encoding in each case except for the primary RMC 0b11 encoding.

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.

Operations	Ideal Exponent
Add	`min(E(FRA), E(FRB))`
Subtract	`min(E(FRA), E(FRB))`
Multiply	`E(FRA) + E(FRB)`
Divide	`E(FRA) - E(FRB)`
Quantize-Immediate	See Instruction Description
Quantize	`E(FRA)`
Reround	See Instruction Description
Round to FP Integer	`max(0, E(FRA))`
Convert to DFP Long	`E(FRA)`
Convert to DFP Extended	`E(FRA)`
Round to DFP Short	`E(FRA)`
Round to DFP Long	`E(FRA)`
Convert from Fixed	`0`
Encode BCD to DPD	`0`
Insert Biased Exponent	`E(FRA)`
Notes: `E(x)` - exponent of the DFP operand in register `x`.	

Figure 5.17: 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 $-\infty$, in which case the sign is negative.
- The sign of the result of the subtract operation x $-$ y is the same as the sign of the result of the add operation x + ($-$y).
- The sign of the result of a multiply or divide operation is the exclusive-OR of the signs of the source operands.

5.5.5 Compare Operations

Two sets of instructions are provided for comparing numerical values: *Compare Ordered* and *Compare Unordered*. In the absence of NaNs, these instructions work the same. These instructions work differently when either of the followings is true:

1. At least one source operand of the instruction is an SNaN and the invalid-operation exception is disabled.
2. When there is no SNaN in any source operand, at least one source operand of the instruction is a QNaN

In case 1, *Compare Unordered* recognizes an invalid-operation exception and sets the `VXSNAN` flag, but *Compare Ordered* recognizes the exception and sets both the `VXSNAN` and `VXVC` flags. In case 2, *Compare Unordered* does not recognize an exception, but *Compare Ordered* recognizes an invalid-operation exception and sets the `VXVC` flag.

For finite numbers, comparisons are performed on values, that is, all redundant forms of a DFP number are treated equal.

Comparisons are always exact and cannot cause an inexact exception.

Comparison ignores the sign of zero, that is, +0 equals -0.

Infinities with like sign compare equal, that is, $+\infty$ equals $+\infty$, and $-\infty$ equals $-\infty$.

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 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 FPCC field and CR field BF.

The *Test Exponent* instruction compares the exponent of the two source operands. The test operation ignores the sign and significand of operands. Infinities compare equal, and NaNs compare equal. The test result is indicated in the FPCC field and CR field BF.

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 FPCC field and CR field BF.

Execution of a test instruction does not cause any DFP exception.

5.5.7 Quantum Adjustment Operations

Four kinds of quantum-adjustment operations are provided: *Quantize*, *Quantize Immediate*, *Reround*, and *Round To FP Integer*. Each of them has an immediate field which specifies whether the rounding mode in FPSCR or a different one is to be used.

The *Quantize* instruction is used to adjust a DFP number to the form that has the specified target exponent. The *Quantize Immediate* instruction is similar to the *Quantize* instruction, except that the target exponent is specified in a 5-bit immediate field as a signed binary integer and has a limited range.

The *Reround* instruction is used to simulate a DFP operation of a precision other than that of DFP Long or DFP Extended. For the *Reround* instruction to produce a result which accurately reflects that which would have resulted from a DFP operation of the desired precision *d* in the range {1: 33} inclusively, the following conditions must be met:

- The precision of the preceding DFP operation must be at least one digit larger than *d*.
- The rounding mode used by the preceding DFP operation must be *round-to-prepare-for-shorter-precision.*

The *Round To FP Integer* instruction is used to round a DFP number to an integer value of the same format. The target exponent is implicitly specified, and is greater than or equal to zero.

5.5.8 Conversion Operations

There are two kinds of conversion operations: data-format conversion and data-type conversion.

5.5.8.1 Data-Format Conversion

The instructions *Convert To DFP Long* and *Convert To DFP Extended* convert DFP operands to wider formats; the instructions *Round To DFP Short* and *Round To DFP Long* convert DFP operands to narrower formats.

When converting a finite number to a wider format, the result is exact. When converting a finite number to a narrower format, the source operand is rounded to the target-format precision, which is specified by the instruction, not by the target register size.

When converting a finite number, the ideal exponent of the result is the source exponent.

Conversion of an infinity or NaN to a different format does not preserve the source combination field. Let N be the width of the target format's combination field.

- When the result is an infinity or a QNaN, the contents of the rightmost N-5 bits of the N-bit target combination field are set to zero.
- When the result is an SNaN, bit 5 of the target format's combination field is set to one and the rightmost N-6 bits of the N-bit target combination field are set to zero.

When converting a NaN to a wider format or when converting an infinity from DFP Short to DFP Long, digits in the source trailing significand field are reencoded using the preferred DPD codes with sufficient zeros appended on the left to form the target trailing significand field. When converting a NaN to a narrower format or when converting an infinity from DFP Long to DFP Short, the appropriate number of leftmost digits of the source trailing significand field are removed and the remaining digits of the field are reencoded using the preferred DPD codes to form the target trailing significand field.

When converting an infinity between DFP Long and DFP Extended, a default infinity with the same sign is produced.

When converting an SNaN between DFP Short and DFP Long, it is converted to an SNaN without causing an invalid-operation exception. When converting an SNaN between DFP Long and DFP Extended, the invalid-operation exception occurs; if the invalid-operation exception is disabled, the result is converted to the corresponding QNaN.

5.5.8.2 Data-Type Conversion

The instructions *Convert From Fixed* and *Convert To Fixed* are provided to convert a number between the DFP data type and the signed 64-bit binary-integer data type.

Conversion of a signed 64-bit binary integer to a DFP Extended number is always exact.

Conversion of a DFP number to a signed 64-bit binary integer results in an invalid-operation exception when the converted value does not fit into the target format, or when the source operand is an infinity or NaN. When the exception is disabled, the most positive integer is returned if the source operand is a positive number or $+\infty$, and the most negative integer is returned if the source operand is a negative number, $-\infty$, or NaN.

5.5.9 Format Operations

The format instructions are provided to facilitate composing or decomposing a DFP number, and consist of *Encode BCD To DPD*, *Decode DPD To BCD*, *Extract Biased Exponent*, *Insert Biased Exponent*, *Shift Significand Left Immediate*, and *Shift Significand Right Immediate*. A source operand of SNaN does not cause an invalid-operation exception, and an SNaN may be produced as the target operand.

5.5.10 DFP Exceptions

This architecture defines the following DFP exceptions:

- Invalid Operation Exception
  - SNaN
  - $\infty - \infty$
  - $\infty \div \infty$
  - $0 \div 0$
  - $\infty \times 0$
  - Invalid Compare
  - Invalid Conversion
- Zero Divide Exception
- Overflow Exception
- Underflow Exception
- Inexact Exception

These exceptions may occur during execution of a DFP instruction.

Each DFP exception, and each category of the Invalid Operation Exception, has an exception status bit in the FPSCR. In addition, each DFP exception has a corresponding enable bit in the FPSCR. The exception status bit indicates occurrence of the corresponding exception. If an exception occurs, the corresponding enable bit governs the result produced by the instruction and, in conjunction with the FE0 and FE1 bits (see the discussion of FE0 and FE1 below), whether and how the system floating-point enabled exception error handler is invoked. (In general, the enabling specified by the enable bit is of invoking the system error handler, not of permitting the exception to occur. The occurrence of an exception depends only on the instruction and its source operands, not on the setting of any control bits. The only deviation from this general rule is that the occurrence of an Underflow Exception may depend on the setting of the enable bit.)

A single instruction, other than ***mtfsfi*** or ***mtfsf***, may set more than one exception bit only in the following cases:

- Inexact Exception may be set with Overflow Exception.
- Inexact Exception may be set with Underflow Exception.
- Invalid Operation Exception (SNaN) may be set with Invalid Operation Exception (Invalid Compare) for *Compare Ordered* instructions
- Invalid Operation Exception (SNaN) may be set with Invalid Operation Exception (Invalid Conversion) for *Convert To Fixed* instructions.

When an exception occurs the instruction execution may be completed or partially completed, depending on the exception and the operation.

For all instructions, except for the Compare and Test instructions, the following exceptions cause the instruction execution to be partially completed. That is, setting of CR field 1(when Rc=1) and exception status flags is performed, but no result is stored into the target FPR or FPR pair. For Compare and Test instructions, instruction execution is always completed, regardless of whether any DFP exception occurs or not, and whether the exception is enabled or not.

- Enabled Invalid Operation
- Enabled Zero Divide

For the remaining kinds of exceptions, instruction execution is completed, a result, if specified by the instruction, is generated and stored into the target FPR or FPR pair, and appropriate status flags are set. The result may be a different value for the enabled and disabled conditions for some of these exceptions. The kinds of exceptions that deliver a result in target FPR are the following:

- Disabled Invalid Operation
- Disabled Zero Divide
- Disabled Overflow
- Disabled Underflow
- Disabled Inexact
- Enabled Overflow
- Enabled Underflow
- Enabled Inexact

Subsequent sections define each of the DFP exceptions and specify the action that is taken when they are detected.

The IEEE standard specifies the handling of exceptional conditions in terms of “traps” and “trap handlers”. In this architecture, a FPSCR exception enable bit of 1 causes generation of the result value specified in the IEEE standard for the “trap enabled” case: the expectation is that the exception will be detected by software, which will revise the result. A FPSCR exception enable bit of 0 causes generation of the “default result” value specified for the

"trap disabled" (or "no trap occurs" or "trap is not implemented") case: the expectation is that the exception will not be detected by software, which will simply use the default result. The result to be delivered in each case for each exception is described in the sections below.

The IEEE default behavior when an exception occurs is to generate a default value and not to notify software. In this architecture, if the IEEE default behavior when an exception occurs is desired for all exceptions, all FPSCR exception enable bits should be set to zero and Ignore Exceptions Mode (see below) should be used. In this case the system floating-point enabled exception error handler is not invoked, even if DFP exceptions occur: software can inspect the FPSCR exception bits if necessary, to determine whether exceptions have occurred.

In this architecture, if software is to be notified that a given kind of exception has occurred, the corresponding FPSCR exception enable bit must be set to one and a mode other than Ignore Exceptions Mode must be used. In this case the system floating-point enabled exception error handler is invoked if an enabled DFP exception occurs. The system floating-point enabled exception error handler is also invoked if a *Move To FPSCR* instruction causes an exception bit and the corresponding enable bit both to be 1; the *Move To FPSCR* instruction is considered to cause the enabled exception.

The FE0 and FE1 bits control whether and how the system floating-point enabled exception error handler is invoked if an enabled DFP exception occurs. The location of these bits and the requirements for altering them are described in Book III, *Power ISA Operating Environment Architecture*. (The system floating-point enabled exception error handler is never invoked because of a disabled DFP exception.) The effects of the four possible settings of these bits are as follows.

FE0	FE1	Description
0	0	**Ignore Exceptions Mode** DFP exceptions do not cause the system floating-point enabled exception error handler to be invoked.
0	1	**Imprecise Nonrecoverable Mode** 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.
1	0	**Imprecise Recoverable Mode** 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.
1	1	**Precise Mode** The system floating-point enabled exception error handler is invoked precisely at the instruction that caused the enabled exception.

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 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. (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 196 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.

Engineering Note

It is permissible for the implementation to be precise in any of the three modes that permit interrupts, or to be recoverable in Nonrecoverable Mode.

5.5.10.1 Invalid Operation Exception

Definition

An Invalid Operation Exception occurs when an operand is invalid for the specified DFP operation. The invalid DFP operations are:

- Any DFP operation on a signaling NaN (SNaN), except for *Test*, *Round To DFP Short*, *Convert To DFP Long*, *Decode DPD To BCD*, *Extract Biased Exponent*, *Insert Biased Exponent*, *Shift Significand Left Immediate*, and *Shift Significand Right Immediate*
- For add or subtract operations, magnitude subtraction of infinities $(+\infty) + (-\infty)$
- 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)
- 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 significance would be greater than x_{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 ***mtfsfi***, ***mtfsf***, or ***mtfsb1*** instruction that sets VXSOFT to 1 (Software Request). The purpose of VXSOFT is to allow software to cause an Invalid Operation Exception for a condition that is not necessarily associated with the execution of a DFP instruction. For example, it might be set by a program that computes a square root, if the source operand is negative.

Action

The action to be taken depends on the setting of the Invalid Operation Exception Enable bit of the FPSCR.

When Invalid Operation Exception is enabled (VE=1) and Invalid Operation occurs, the following actions are taken:

1. One or two Invalid Operation Exceptions are set:

VXSNAN	(if SNaN)
VXISI	(if $\infty - \infty$)
VXIDI	(if $\infty \div \infty$)
VXZDZ	(if $0 \div 0$)
VXIMZ	(if $\infty \times 0$)
VXVC	(if invalid comp)
VXCVI	(if invalid conversion)

2. If the operation is an arithmetic, quantum-adjustment, conversion, or format,
   the target FPR is unchanged,
   FR and FI are set to zero, and
   FPRF is unchanged.
3. If the operation is a compare,
   FR, FI, and C are unchanged, and
   FPCC is set to reflect unordered.

When Invalid Operation Exception is disabled (VE=0) and Invalid Operation occurs, the following actions are taken:

1. One or two Invalid Operation Exceptions are set:

VXSNAN	(if SNaN)
VXISI	(if $\infty - \infty$)
VXIDI	(if $\infty \div \infty$)
VXZDZ	(if $0 \div 0$)
VXIMZ	(if $\infty \times 0$)
VXVC	(if invalid comp)
VXCVI	(if invalid conversion)

2. If the operation is an arithmetic, quantum-adjustment, *Round to DFP Long*, *Convert to DFP Extended*, or format
   the target FPR is set to a Quiet NaN
   FR and FI are set to zero
   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.
   FR and FI are set to zero
   FPRF is unchanged
4. If the operation is a compare,
   FR, FI, and C are unchanged
   FPCC is set to reflect unordered

5.5.10.2 Zero Divide Exception

Definition

A Zero Divide Exception occurs when a Divide instruction is executed with a zero divisor value and a finite nonzero dividend value.

Action

The action to be taken depends on the setting of the Zero Divide Exception Enable bit of the FPSCR.

When Zero Divide Exception is enabled (ZE=1) and Zero Divide occurs, the following actions are taken:

1. Zero Divide Exception is set
   ZX $\leftarrow$ 1
2. The target FPR is unchanged
3. FR and FI are set to zero
4. FPRF is unchanged

When Zero Divide Exception is disabled (ZE=0) and Zero Divide occurs, the following actions are taken:

1. Zero Divide Exception is set
   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. FR and FI are set to zero
4. 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 *Reround*, the following describes the handling of the IEEE overflow exception condition. The *Reround* operation does not recognize an overflow exception condition.

The action to be taken depends on the setting of the Overflow Exception Enable bit of the FPSCR.

When Overflow Exception is enabled (OE=1) and overflow occurs, the following actions are taken:

1. Overflow Exception is set
   OX $\leftarrow$ 1
2. The infinitely precise result is divided by 10^{α}. That is, the exponent adjustment α is subtracted from the exponent. This is called the *wrapped result*. The exponent adjustment for all operations, except for *Round To DFP Short* and *Round To DFP Long*, is 576 for DFP Long and 9216 for DFP Extended. For *Round To DFP Short* and *Round To DFP Long*, the exponent adjustment is 192 for the source format of DFP Long and 3072 for the source format of DFP Extended.
3. The wrapped result is rounded to the target-format precision. This is called the *wrapped rounded result*.
4. If the wrapped rounded result has only one form, it is the delivered result. If the wrapped rounded result has redundant forms and is exact, the result of the form that has the exponent closest to the wrapped ideal exponent is returned. If the wrapped rounded result has redundant forms and is inexact, the result of the form that has the smallest exponent is returned. The wrapped ideal exponent is the result of subtracting the exponent adjustment from the ideal exponent.
5. FPRF is set to indicate the class and sign of the result ($\pm$ Normal Number)

When Overflow Exception is disabled (OE=0) and overflow occurs, the following actions are taken:

1. Overflow Exception is set
   OX $\leftarrow$ 1
2. Inexact Exception is set
   XX $\leftarrow$ 1

3. The result is determined by the rounding mode and the sign of the intermediate result as follows.

Rounding Mode	Sign of intermediate result	
	Plus	Minus
Round to Nearest, Ties to Even	$+\infty$	$-\infty$
Round toward 0	$+\mathrm{N}_{max}$	$-\mathrm{N}_{max}$
Round toward $+\infty$	$+\infty$	$-\mathrm{N}_{max}$
Round toward $-\infty$	$+\mathrm{N}_{max}$	$-\infty$
Round to Nearest, Ties away from 0	$+\infty$	$-\infty$
Round to Nearest, Ties toward 0	$+\infty$	$-\infty$
Round away from 0	$+\infty$	$-\infty$
Round to prepare for shorter precision	$+\mathrm{N}_{max}$	$-\mathrm{N}_{max}$

Figure 5.18: Overflow Results When Exception Is Disabled

4. The result is placed into the target FPR
5. FR is set to one if the returned result is $\pm\infty$, and is set to zero if the returned result is $\pm\mathrm{N}_{max}$
6. FI is set to one
7. FPRF is set to indicate the class and sign of the result ($\pm\infty$ or $\pm$ Normal number)

5.5.10.4 Underflow Exception

Definition

Except for *Reround*, the following describes the handling of the IEEE underflow exception condition. The *Reround* operation does not recognize an underflow exception condition.

The Underflow Exception is defined differently for the enabled and disabled states. However, a tininess condition is recognized in both states when a result computed as though both the precision and exponent range were unbounded would be nonzero and less than the target format's smallest normal number, N_{min}, in magnitude.

Unless otherwise defined in the instruction description, an underflow exception occurs as follows:

- Enabled:
  When the tininess condition is recognized.
- Disabled:
  When the tininess condition is recognized and when the delivered result value differs from what would have been computed were both the precision and the exponent range unbounded.

Action

The action to be taken depends on the setting of the Underflow Exception Enable bit of the FPSCR.

When Underflow Exception is enabled (UE=1) and underflow occurs, the following actions are taken:

1. Underflow Exception is set
   $\mathrm{UX} \leftarrow 1$
2. The infinitely precise result is multiplied by 10^{α}. That is, the exponent adjustment α is added to the exponent. This is called the *wrapped result*. The exponent adjustment for all operations, except for *Round To DFP Short* and *Round To DFP Long*, is 576 for DFP Long and 9216 for DFP Extended. For *Round To DFP Short* and *Round To DFP Long*, the exponent adjustment is 192 for the source format of DFP Long and 3072 for the source format of DFP Extended.
3. The wrapped result is rounded to the target-format precision. This is called the *wrapped rounded result*.
4. If the wrapped rounded result has only one form, it is the delivered result. If the wrapped rounded result has redundant forms and is exact, the result of the form that has the exponent closest to the wrapped ideal exponent is returned. If the wrapped rounded result has redundant forms and is inexact, the result of the form that has the smallest exponent is returned. The wrapped ideal exponent is the result of adding the exponent adjustment to the ideal exponent.
5. FPRF is set to indicate the class and sign of the result ($\pm$ Normal number)

When Underflow Exception is disabled (UE=0) and underflow occurs, the following actions are taken:

1. Underflow Exception is set
   $\mathrm{UX} \leftarrow 1$
1. The infinitely precise result is rounded to the target-format precision.
2. 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.
3. FPRF is set to indicate the class and sign of the result ($\pm$ Normal number, $\pm$ Subnormal Number, or $\pm$ Zero)

5.5.10.5 Inexact Exception

Definition

Except for *Round to FP Integer Without Inexact*, the following describes the handling of the IEEE inexact exception condition. The *Round to FP Integer Without Inexact* does not recognize an inexact exception condition.

An Inexact Exception occurs when either of two conditions occur during rounding:

1. The delivered result differs from what would have been computed were both the precision and exponent range unbounded.
2. The rounded result overflows and Overflow Exception is disabled.

Action

The action to be taken does not depend on the setting of the Inexact Exception Enable bit of the FPSCR.

When Inexact Exception occurs, the following actions are taken:

1. Inexact Exception is set
   $XX \leftarrow 1$
2. The rounded or overflowed result is placed into the target FPR
3. FPRF is set to indicate the class and sign of the result

Programming Note

In some implementations, enabling Inexact Exceptions may degrade performance more than does enabling other types of floating-point exception.

5.5.11 Summary of Normal Rounding And Range Actions

Figure 5.19 and Figure 5.20 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.

Range of v	Case	Result (r) when Rounding Mode Is							
		RNE	RNTZ	RNAZ	RAFZ	RTMI	RFSP	RTPI	RTZ
$v < -Nmax$, $q < -Nmax$	Overflow	$-\infty^1$	$-\infty^1$	$-\infty^1$	$-\infty^1$	$-\infty^1$	$-Nmax$	$-Nmax$	$-Nmax$
$v < -Nmax$, $q = -Nmax$	Normal	$-Nmax$	$-Nmax$	$-Nmax$	–	–	$-Nmax$	$-Nmax$	$-Nmax$
$-Nmax \le v \le -Nmin$	Normal	b	b	b	b	b	b	b	b
$-Nmin < v \le -Dmin$	Tiny	b*	b*	b*	b*	b*	b*	b	b
$-Dmin < v < -Dmin/2$	Tiny	$-Dmin$	$-Dmin$	$-Dmin$	$-Dmin$	$-Dmin$	$-Dmin$	-0	-0
$v = -Dmin/2$	Tiny	-0	-0	$-Dmin$	$-Dmin$	$-Dmin$	$-Dmin$	-0	-0
$-Dmin/2 < v < 0$	Tiny	-0	-0	-0	$-Dmin$	$-Dmin$	$-Dmin$	-0	-0
$v = 0$	EZD	+0	+0	+0	+0	-0	+0	+0	+0
$0 < v < +Dmin/2$	Tiny	+0	+0	+0	+Dmin	+0	+Dmin	+Dmin	+0
$v = +Dmin/2$	Tiny	+0	+0	+Dmin	+Dmin	+0	+Dmin	+Dmin	+0
$+Dmin/2 < v < +Dmin$	Tiny	+Dmin	+Dmin	+Dmin	+Dmin	+0	+Dmin	+Dmin	+0
$+Dmin \le v < +Nmin$	Tiny	b*	b*	b*	b*	b	b*	b*	b
$+Nmin \le v \le +Nmax$	Normal	b	b	b	b	b	b	b	b
$+Nmax < v$, $q = +Nmax$	Normal	+Nmax	+Nmax	+Nmax	–	+Nmax	+Nmax	–	+Nmax
$+Nmax < v$, $q > +Nmax$	Overflow	$+\infty^1$	$+\infty^1$	$+\infty^1$	$+\infty^1$	+Nmax	+Nmax	$+\infty^1$	+Nmax

Explanation:

–	This situation cannot occur.
1	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.
b	The value derived when the precise result v is rounded to the destination's precision, including both bounded precision and bounded exponent range.
q	The value derived when the precise result v is rounded to the destination's precision, but assuming an unbounded exponent range.
r	This is the returned value when neither overflow nor underflow is enabled.
v	Precise result before rounding, assuming unbounded precision and an unbounded exponent range. For data-format conversion operations, v is the source value.
Dmin	Smallest (in magnitude) representable subnormal number in the arget format.
EZD	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.)
Nmax	Largest (in magnitude) representable finite number in the target format.
Nmin	Smallest (in magnitude) representable normalized number in the target format.
RAFZ	Round away from 0.
RFSP	Round to Prepare for Shorter Precision.
RNAZ	Round to Nearest, Ties away from 0.
RNE	Round to Nearest, Ties to even.
RNTZ	Round to Nearest, Ties toward 0.
RTPI	Round toward $+\infty$.
RTMI	Round toward $-\infty$.
RTZ	Round toward 0.

Figure 5.19: Rounding and Range Actions (Part 1)

Case	Is r inexact (r≠v)	OE=1	UE=1	XE=1	Is r Incremented (|r|>|v|)	Is q inexact (q≠v)	Is q Incremented (|q|>|v|)	Returned Results and Status Setting*
Overflow	Yes[1]	No	–	No	No	–	–	T(r), OX←1, FI←1, FR←0, XX←1
Overflow	Yes[1]	No	–	No	Yes	–	–	T(r), OX←1, FI←1, FR←1, XX←1
Overflow	Yes[1]	No	–	Yes	No	–	–	T(r), OX←1, FI←1, FR←0, XX←1, TX
Overflow	Yes[1]	No	–	Yes	Yes	–	–	T(r), OX←1, FI←1, FR←1, XX←1, TX
Overflow	Yes[1]	Yes	–	–	–	No	No[1]	Tw(q÷β), OX←1, FI←0, FR←0, TO
Overflow	Yes[1]	Yes	–	–	–	Yes	No	Tw(q÷β), OX←1, FI←1, FR←0, XX←1, TO
Overflow	Yes[1]	Yes	–	–	–	Yes	Yes	Tw(q÷β), OX←1, FI←1, FR←1, XX←1, TO
Normal	No	–	–	–	–	–	–	T(r), FI←0, FR←0
Normal	Yes	–	–	No	No	–	–	T(r), FI←1, FR←0, XX←1
Normal	Yes	–	–	No	Yes	–	–	T(r), FI←1, FR←1, XX←1
Normal	Yes	–	–	Yes	No	–	–	T(r), FI←1, FR←0, XX←1, TX
Normal	Yes	–	–	Yes	Yes	–	–	T(r), FI←1, FR←1, XX←1, TX
Tiny	No	–	No	–	–	–	–	T(r), FI←0, FR←0
Tiny	No	–	Yes	–	–	No[1]	No[1]	Tw(q×β), UX←1, FI←0, FR← 0, TU
Tiny	Yes	–	No	No	No	–	–	T(r), UX←1, FI←1, FR←0, XX←1
Tiny	Yes	–	No	No	Yes	–	–	T(r), UX←1, FI←1, FR←1, XX←1
Tiny	Yes	–	No	Yes	No	–	–	T(r), UX←1, FI←1, FR←0, XX←1, TX
Tiny	Yes	–	No	Yes	Yes	–	–	T(r), UX←1, FI←1, FR←1, XX←1, TX
Tiny	Yes	–	Yes	–	–	No	No[1]	Tw(q×β), UX←1, FI←0, FR←0, TU
Tiny	Yes	–	Yes	–	–	Yes	No	Tw(q×β), UX←1, FI←1, FR←0, XX←1, TU
Tiny	Yes	–	Yes	–	–	Yes	Yes	Tw(q×β), UX←1, FI←1, FR←1, XX←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.

* Rounding sets only FI and FR. Setting of OX, XX, or UX is part of the exception actions. They are listed here for reference.

β Wrap adjust, which depends on the type of operation and operand format. For all operations except *Round to DFP Short* and *Round to DFP Long*, the wrap adjust depends on the target format: $\beta = 10^{\alpha}$, where α is 576 for DFP Long, and 9216 for DFP Extended. For *Round to DFP Short* and *Round to DFP Long*, the wrap adjust depends on the source format: $\beta = 10^{\kappa}$ where κ is 192 for DFP Long and 3072 for DFP Extended.

q The value derived when the precise result v is rounded to destination's precision, but assuming an unbounded exponent range.

r The result as defined in Part 1 of this figure.

v Precise result before rounding, assuming unbounded precision and unbounded exponent range.

FI Floating-Point-Fraction-Inexact status flag, FI. This status flag is non-sticky.

FR Floating-Point-Fraction-Rounded status flag, FR.

OX Floating-Point Overflow Exception status flag, OX.

TO The system floating-point enabled exception error handler is invoked for the overflow exception if FE0 and FE1 are set to any mode other than the ignore-exception mode.

TU The system floating-point enabled exception error handler is invoked for the underflow exception if FE0 and FE1 are set to any mode other than the ignore-exception mode.

TX The system floating-point enabled exception error handler is invoked for the inexact exception if FE0 and FE1 are set to any mode other than the ignore-exception mode.

T(x) The value x is placed at the target operand location.

Tw(x) The wrapped rounded result x is placed at the target operand location. For all operations except data format conversions, the wrapped rounded result is in the same format and length as normal results at the target location. For data format conversions, the wrapped rounded result is in the same format and length as the source, but rounded to the target-format precision.

UX Floating-Point-Underflow-Exception status bit.

XX Float-Point Inexact exception status bit. The flag is a sticky version of FI. When FI is set to a new value, the new value of XX is set to the result of ORing the old value of XX with the new value of FI.

Figure 5.20: 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, FRXp denotes a FPR pair and must address an even-odd pair. If the FRXp field specifies an odd-numbered register, then the instruction form is invalid. The notation FRX[p] means either a FPR, FRX, or a FPR pair, FRXp.

For DFP instructions, if a DFP operand is returned, the trailing significand field of the target operand is encoded using preferred DPD codes.

5.6.1 DFP Arithmetic Instructions

All DFP arithmetic instructions are X-form instructions. They all set the FI and FR status flags, and also set the FPRF field. Furthermore, they all have an ideal exponent assigned and employ the record bit (Rc).

The arithmetic instructions consist of Add, Divide, Multiply, and Subtract.

Engineering Note

DFP uses three instruction formats: X-form, Z22-form, and Z23-form. This note refers to the latter two formats as "Z-form".

All Z-form DFP instructions differ from the X-form instruction only in the starting point of the extended-opcode field. The Z-form instructions have an immediate operand which takes one or two leftmost bits of the X-form extended-opcode field.

Even though these DFP instructions are defined to be Z-form architecturally, they should be interpreted as X-form instructions by the decoder to simplify hardware design. The opcodes of these Z-form instructions were assigned in such a way that they all map into the X-form opcode space.

DFP Add X-form

dadd	FRT,FRA,FRB	(Rc=0)
dadd.	FRT,FRA,FRB	(Rc=1)

59	FRT	FRA	FRB	2	Rc
0	6	11	16	21	31

DFP Add Quad X-form

daddq	FRTp,FRAp,FRBp	(Rc=0)
daddq.	FRTp,FRAp,FRBp	(Rc=1)

63	FRTp	FRAp	FRBp	2	Rc
0	6	11	16	21	31

The DFP operand in FRA[p] is added to the DFP operand in FRB[p].

The result is rounded to the target-format precision under control of DRN (bits 29:31 of the FPSCR). An appropriate form of the rounded result is selected based on the ideal exponent and is placed in FRT[p]. The ideal exponent is the smaller exponent of the two source operands.

Figure 5.21 summarizes the actions for Add. Figure 5.21 does not include the setting of FPRF. FPRF is always set to the class and sign of the result, except for an enabled invalid-operation exception, in which case the field remains unchanged.

dadd[***q***][***.***] are treated as *Floating-Point* instructions in terms of resource availability.

Special Registers Altered:

Register	*Field(s)*	
FPSCR	FPRF FR FI FX OX UX XX VXSNAN VXISI	
CR	CR1	(if Rc=1)

DFP Subtract X-form

dsub	FRT,FRA,FRB	(Rc=0)
dsub.	FRT,FRA,FRB	(Rc=1)

59	FRT	FRA	FRB	514	Rc
0	6	11	16	21	31

DFP Subtract Quad X-form

dsubq	FRTp,FRAp,FRBp	(Rc=0)
dsubq.	FRTp,FRAp,FRBp	(Rc=1)

63	FRTp	FRAp	FRBp	514	Rc
0	6	11	16	21	31

The DFP operand in FRB[p] is subtracted from the DFP operand in FRA[p].

The result is rounded to the target-format precision under control of DRN (bits 29:31 of the FPSCR). An appropriate form of the rounded result is selected based on the ideal exponent and is placed in FRT[p]. The ideal exponent is the smaller exponent of the two source operands.

The execution of Subtract is identical to that of Add, except that the operand in FRB participates in the operation with its sign bit inverted. See Figure 5.21. The table does not include the setting of FPRF. FPRF is always set to the class and sign of the result, except for an enabled invalid-operation exception, in which case the field remains unchanged.

dsub[***q***][***.***] are treated as *Floating-Point* instructions in terms of resource availability.

Special Registers Altered:

Register	*Field(s)*	
FPSCR	FPRF FR FI FX OX UX XX VXSNAN VXISI	
CR	CR1	(if Rc=1)

Operand a in FRA[p] is	Actions for Add (a + b) when operand b in FRB[p] is				
	$-\infty$	F	$+\infty$	QNaN	SNaN
$-\infty$	T(−dINF)	T(−dINF)	VXISI: T(dNaN)	P(b)	VXSNAN: U(b)
F	T(−dINF)	S(a+b)	T(+dINF)	P(b)	VXSNAN: U(b)
$+\infty$	VXISI: T(dNaN)	T(+dINF)	T(+dINF)	P(b)	VXSNAN: U(b)
QNaN	P(a)	P(a)	P(a)	P(a)	VXSNAN: U(b)
SNaN	VXSNAN: U(a)	VXSNAN: U(a)	VXSNAN: U(a)	VXSNAN: U(a)	VXSNAN: U(a)

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 202)
- +dINF — Default plus infinity.
- −dINF — Default minus infinity.
- dNaN — Default quiet NaN.
- F — All finite numbers, including zeros.
- P(x) — The QNaN of operand x is propagated and placed in FRT[p].
- S(x) — The value x is placed in FRT[p] with the sign set by the rules of algebra. When the source operands have the same sign, the sign of the result is the same as the sign of the operands, including the case when the result is zero. When the operands have opposite signs, the sign of a zero result is positive in all rounding modes, except round toward $-\infty$, in which case, the sign is minus.
- T(x) — The value x is placed in FRT[p].
- U(x) — The SNaN of operand x is converted to the corresponding QNaN and placed in FRT[p].
- VXISI: — Floating-Point Invalid Operation (Infinity − Infinity) exception occurs. The result is produced only when the exception is disabled. (See Section 5.5.10.1 “Invalid Operation Exception” on page 198.)
- VXSNAN: — Floating-Point Invalid Operation (SNaN) exception occurs. The result is produced only when the exception is disabled. (See Section 5.5.10.1 “Invalid Operation Exception” on page 198.)

Figure 5.21: Actions: Add

DFP Multiply X-form

dmul	FRT,FRA,FRB	(Rc=0)
dmul.	FRT,FRA,FRB	(Rc=1)

59	FRT	FRA	FRB	34	Rc
0	6	11	16	21	31

DFP Multiply Quad X-form

dmulq	FRTp,FRAp,FRBp	(Rc=0)
dmulq.	FRTp,FRAp,FRBp	(Rc=1)

63	FRTp	FRAp	FRBp	34	Rc
0	6	11	16	21	31

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 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 5.22 summarizes the actions for Multiply. Figure 5.22 does not include the setting of FPRF. FPRF is always set to the class and sign of the result, except for an enabled invalid-operation exception, in which case the field remains unchanged.

dmul[***q***][***.***] are treated as *Floating-Point* instructions in terms of resource availability.

Special Registers Altered:

Register	***Field(s)***	
FPSCR	FPRF FR FI FX OX UX XX VXSNAN VXIMZ	
CR	CR1	(if Rc=1)

Operand a in FRA[p] is	Actions for Multiply (a×b) when operand b in FRB[p] is				
	0	**Fn**	**∞**	**QNaN**	**SNaN**
0	S(a × b)	S(a × b)	VXIMZ: T(dNaN)	P(b)	VXSNAN: U(b)
Fn	S(a × b)	S(a × b)	S(dINF)	P(b)	VXSNAN: U(b)
∞	VXIMZ: T(dNaN)	S(dINF)	S(dINF)	P(b)	VXSNAN: U(b)
QNaN	P(a)	P(a)	P(a)	P(a)	VXSNAN: U(b)
SNaN	VXSNAN: U(a)	VXSNAN: U(a)	VXSNAN: U(a)	VXSNAN: U(a)	VXSNAN: U(a)

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 202)
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].
VXIMZ:	Floating-Point Invalid Operation (Infinity × Zero) exception occurs. The result is produced only when the exception is disabled. (See Section 5.5.10.1 “Invalid Operation Exception” on page 198.)
VXSNAN:	Floating-Point Invalid Operation (SNaN) exception occurs. The result is produced only when the exception is disabled. (See Section 5.5.10.1 “Invalid Operation Exception” on page 198.)

Figure 5.22: Actions: Multiply

DFP Divide X-form

ddiv	FRT,FRA,FRB	(Rc=0)
ddiv.	FRT,FRA,FRB	(Rc=1)

59	FRT	FRA	FRB	546	Rc
0	6	11	16	21	31

DFP Divide Quad X-form

ddivq	FRTp,FRAp,FRBp	(Rc=0)
ddivq.	FRTp,FRAp,FRBp	(Rc=1)

63	FRTp	FRAp	FRBp	546	Rc
0	6	11	16	21	31

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 5.23 summarizes the actions for Divide. Figure 5.23 does not include the setting of FPRF. FPRF is always set to the class and sign of the result, except for an enabled invalid-operation and enabled zero-divide exceptions, in which cases the field remains unchanged.

***ddiv*[*q*][*.*]** are treated as *Floating-Point* instructions in terms of resource availability.

Special Registers Altered:

Register	*Field(s)*	
FPSCR	FPRF FR FI FX OX UX ZX XX VXSNAN VXIDI VXZDZ	
CR	CR1	(if Rc=1)

Operand a in FRA[p] is	Actions for Divide (a ÷ b) when operand b in FRB[p] is				
	0	**Fn**	**∞**	**QNaN**	**SNaN**
0	VXZDZ: T(dNaN)	S(a÷b)	S(zt)	P(b)	VXSNAN: U(b)
Fn	Zx: S(dINF)	S(a÷b)	S(zt)	P(b)	VXSNAN: U(b)
∞	S(dINF)	S(dINF)	VXIDI: T(dNaN)	P(b)	VXSNAN: U(b)
QNaN	P(a)	P(a)	P(a)	P(a)	VXSNAN: U(b)
SNaN	VXSNAN: U(a)	VXSNAN: U(a)	VXSNAN: U(a)	VXSNAN: U(a)	VXSNAN: U(a)

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 202.)
- 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: — Floating-Point Invalid Operation (Infinity ÷ Infinity) exception occurs. The result is produced only when the exception is disabled. (See Section 5.5.10.1 “Invalid Operation Exception” on page 198 for the exception actions.)
- VXSNAN: — Floating-Point Invalid Operation (SNaN) exception occurs. The result is produced only when the exception is disabled. (See Section 5.5.10.1 “Invalid Operation Exception” on page 198.)
- VXZDZ: — Floating-Point Invalid Operation (Zero ÷ Zero) exception occurs. The result is produced only when the exception is disabled. (See Section 5.5.10.1 “Invalid Operation Exception” on page 198.)
- 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 199.)

Figure 5.23: Actions: Divide

5.6.2 DFP Compare Instructions

The DFP compare instructions consist of the *Compare Ordered* and *Compare Unordered* instructions. The compare instructions do not provide the record bit.

The comparison sets the designated CR field to indicate the result. FPCC is set in the same way.

The codes in the CR field BF and FPCC are defined for the DFP compare operations as follows.

Bit	Name	Description
0	FL	(FRA[p]) < (FRB[p])
1	FG	(FRA[p]) > (FRB[p])
2	FE	(FRA[p]) = (FRB[p])
3	FU	(FRA[p]) ? (FRB[p])

DFP Compare Unordered X-form

dcmpu BF,FRA,FRB

59	BF	//	FRA	FRB	642	/
0	6	9	11	16	21	31

DFP Compare Unordered Quad X-form

dcmpuq BF,FRAp,FRBp

63	BF	//	FRAp	FRBp	642	/
0	6	9	11	16	21	31

The DFP operand in FRA[p] is compared to the DFP operand in FRB[p]. The result of the compare is placed into CR field BF and the FPCC.

***dcmpu*[*q*]** are treated as *Floating-Point* instructions in terms of resource availability.

Special Registers Altered:

Register	*Field(s)*
CR	field BF
FPSCR	FPCC FX VXSNAN

Operand a **in** FRA[p] **is**	**Actions for Compare Unordered** (a:b) **when operand b in** FRB[p] **is**				
	−∞	F	+∞	QNaN	SNaN
−∞	AeqB	AltB	AltB	AuoB	Fu, VXSNAN
F	AgtB	C(a:b)	AltB	AuoB	Fu, VXSNAN
+∞	AgtB	AgtB	AeqB	AuoB	Fu, VXSNAN
QNaN	AuoB	AuoB	AuoB	AuoB	Fu, VXSNAN
SNaN	Fu, VXSNAN	Fu, VXSNAN	Fu, VXSNAN	Fu, VXSNAN	Fu, VXSNAN

Explanation:

C(a:b)	Algebraic comparison. See the table below.
F	All finite numbers, including zeros.
AeqB	CR field BF and FPCC are set to 0b0010.
AgtB	CR field BF and FPCC are set to 0b0100.
AltB	CR field BF and FPCC are set to 0b1000.
AuoB	CR field BF and FPCC are set to 0b0001.
VXSNAN	Floating-Point Invalid Operation (SNaN) exception occurs. (See Section 5.5.10.1 “Invalid Operation Exception” on page 198.)

Relation of Value a **to Value** b	**Action for** C(a:b)
a = b	AeqB
a < b	AltB
a > b	AgtB

Figure 5.24: Actions: Compare Unordered

DFP Compare Ordered X-form

dcmpo BF,FRA,FRB

59	BF	//	FRA	FRB	130	/
0	6	9	11	16	21	31

DFP Compare Ordered Quad X-form

dcmpoq BF,FRAp,FRBp

63	BF	//	FRAp	FRBp	130	/
0	6	9	11	16	21	31

The DFP operand in FRA[p] is compared to the DFP operand in FRB[p]. The result of the compare is placed into CR field BF and the FPCC.

dcmpo[***q***] are treated as *Floating-Point* instructions in terms of resource availability.

Special Registers Altered:

Register	***Field(s)***
CR	field BF
FPSCR	FPCC FX VXSNAN VXVC

Operand a in FRA[p] is	**Actions for Compare ordered (a:b) when operand b in FRB[p] is**				
	−∞	F	+∞	QNaN	SNaN
−∞	AeqB	AltB	AltB	AuoB, VXVC	AuoB, VXSV
F	AgtB	C(a:b)	AltB	AuoB, VXVC	AuoB, VXSV
+∞	AgtB	AgtB	AeqB	AuoB, VXVC	AuoB, VXSV
QNaN	AuoB, VXVC	AuoB, VXVC	AuoB, VXVC	AuoB, VXVC	AuoB, VXSV
SNaN	AuoB, VXSV	AuoB, VXSV	AuoB, VXSV	AuoB, VXSV	AuoB, VXSV

Explanation:

C(a:b)	Algebraic comparison. See the table below
F	All finite numbers, including zeros
AeqB	CR field BF and FPCC are set to 0b0010.
AgtB	CR field BF and FPCC are set to 0b0100.
AltB	CR field BF and FPCC are set to 0b1000.
AuoB	CR field BF and FPCC are set to 0b0001.
VXSV	Floating-Point Invalid Operation (SNaN) exception occurs. Additionally, if the exception is disabled (VE=0), then VXVC is also set to one. (See Section 5.5.10.1 “Invalid Operation Exception” on page 198.)
VXVC	Floating-Point Invalid Operation (Invalid Compare) exception occurs. (See Section 5.5.10.1 “Invalid Operation Exception” on page 198.)

Relation of Value a to Value b	**Action for C(a:b)**
a = b	AeqB
a < b	AltB
a > b	AgtB

Figure 5.25: Actions: Compare Ordered

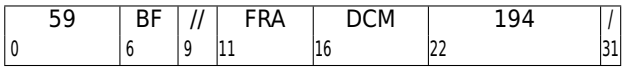
5.6.3 DFP Test Instructions

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 $FPSCR_{FPCC}$ is set in the same way.

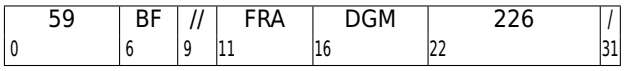
DFP Test Data Class Z22-form

dtstdc BF,FRA,DCM



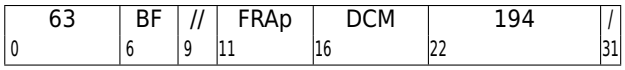
DFP Test Data Group Z22-form

dtstdg BF,FRA,DGM



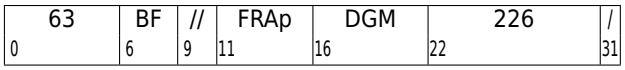
DFP Test Data Class Quad Z22-form

dtstdcq BF,FRAp,DCM



DFP Test Data Group Quad Z22-form

dtstdgq BF,FRAp,DGM



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 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

dtstdc[***q***] are treated as *Floating-Point* instructions in terms of resource availability.

Special Registers Altered:

Register	*Field(s)*
CR	field BF
FPSCR	FPCC

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_{max}, or the minimum exponent, x_{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 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

dtstdg[***q***] are treated as *Floating-Point* instructions in terms of resource availability.

Special Registers Altered:

Register	*Field(s)*
CR	field BF
FPSCR	FPCC

DFP Test Exponent X-form

dtstex BF,FRA,FRB

59	BF	//	FRA	FRB	162	/
0	6	9	11	16	21	31

DFP Test Exponent Quad X-form

dtstexq BF,FRAp,FRBp

63	BF	//	FRAp	FRBp	162	/
0	6	9	11	16	21	31

The exponent value (`Ea`) of the DFP operand in `FRA[p]` is compared to the exponent value (`Eb`) of the DFP operand in `FRB[p]`. The result of the compare is placed into CR field `BF` and the `FPCC`.

The codes in the CR field `BF` and `FPCC` are defined for the *DFP Test Exponent* operations as follows.

Bit	Description
0	`Ea < Eb`
1	`Ea > Eb`
2	`Ea = Eb`
3	`Ea ? Eb`

dtstex[***q***] are treated as *Floating-Point* instructions in terms of resource availability.

Special Registers Altered:

Register	*Field(s)*
CR	field BF
FPSCR	FPCC

Operand a in FRA[p] is	Actions for Test Exponent (`Ea:Eb`) when operand `b` in `FRB[p]` is			
	F	∞	QNaN	SNaN
F	`C(Ea:Eb)`	`AuoB`	`AuoB`	`AuoB`
∞	`AuoB`	`AeqB`	`AuoB`	`AuoB`
QNaN	`AuoB`	`AuoB`	`AeqB`	`AeqB`
SNaN	`AuoB`	`AuoB`	`AeqB`	`AeqB`

Explanation:

`C(Ea:Eb)`	Algebraic comparison. See the table below.
F	All finite numbers, including zeros
`AeqB`	CR field `BF` and `FPCC` are set to `0b0010`.
`AgtB`	CR field `BF` and `FPCC` are set to `0b0100`.
`AltB`	CR field `BF` and `FPCC` are set to `0b1000`.
`AuoB`	CR field `BF` and `FPCC` are set to `0b0001`.

Relation of Value `Ea` to Value `Eb`	Action for `C(Ea:Eb)`
`Ea = Eb`	`AeqB`
`Ea < Eb`	`AltB`
`Ea > Eb`	`AgtB`

Figure 5.26: Actions: Test Exponent

DFP Test Significance X-form

dtstsf BF,FRA,FRB

59	BF	//	FRA	FRB	674	/
0	6	9	11	16	21	31

DFP Test Significance Quad X-form

dtstsfq BF,FRA,FRBp

63	BF	//	FRA	FRBp	674	/
0	6	9	11	16	21	31

Let k be the contents of bits 58:63 of FPR[FRA] that specifies the reference significance.

For *dtstsf*, let the value NSDb be the number of significant digits of the DFP value in FPR[FRB].

For *dtstsfq*, let the value NSDb be the number of significant digits of the DFP value in FPR[FRBp:FRBp+1].

For this instruction, the number of significant digits of the value 0 is considered to be zero.

NSDb is compared to k. The result of the compare is placed into CR field BF and the FPCC as follows.

Bit	Description
0	k $\neq$ 0 and k < NSDb
1	k $\neq$ 0 and k > NSDb, or k = 0
2	k $\neq$ 0 and k = NSDb
3	k ? NSDb

***dtstsf*[*q*]** are treated as *Floating-Point* instructions in terms of resource availability.

Special Registers Altered:

Register	*Field(s)*
CR	field BF
FPSCR	FPCC

Programming Note

The reference significance can be loaded into a FPR using a *Load Float as Integer Word Algebraic* instruction

Actions for Test Significance when the operand in VSR[FRB] or VSR[FRBp:FRBp+1] is			
F	**∞**	**QNaN**	**SNaN**
C(UIM:NSDb)	AuoB	AuoB	AuoB

Explanation:

C(k:NSDb)	Algebraic comparison. See the table below.
F	All finite numbers, including zeros.
AeqB	CR field BF and FPCC are set to 0b0010.
AgtB	CR field BF and FPCC are set to 0b0100.
AltB	CR field BF and FPCC are set to 0b1000.
AuoB	CR field BF and FPCC are set to 0b0001.

Relation of Value NSDb to Value k	Action for C(k:NSDb)
k $\neq$ 0 and k = NSDb	AeqB
k $\neq$ 0 and k < NSDb	AltB
k $\neq$ 0 and k > NSDb, or k = 0	AgtB

Figure 5.27: Actions: Test Significance

DFP Test Significance Immediate X-form

dtstsfi BF,UIM,FRB

59	BF	/	UIM	FRB	675	/
0	6	9	10	16	21	31

DFP Test Significance Immediate Quad X-form

dtstsfiq BF,UIM,FRBp

63	BF	/	UIM	FRBp	675	/
0	6	9	10	16	21	31

Let the value UIM specify the reference significance.

For *dtstsfi*, let the value NSDb be the number of significant digits of the DFP value in FPR[FRB].

For *dtstsfiq*, let the value NSDb be the number of significant digits of the DFP value in FPR[FRBp:FRBp+1].

For this instruction, the number of significant digits of the value 0 is considered to be zero.

NSDb is compared to UIM. The result of the compare is placed into CR field BF and the FPCC as follows.

Bit	Description
0	UIM $\neq$ 0 and UIM < NSDb
1	UIM $\neq$ 0 and UIM > NSDb, or UIM = 0
2	UIM $\neq$ 0 and UIM = NSDb
3	UIM ? NSDb

dtstsfi[***q***] are treated as *Floating-Point* instructions in terms of resource availability.

Special Registers Altered:

Register	*Field(s)*
CR	field BF
FPSCR	FPCC

Actions for Test Significance when the operand in VSR[FRB] or VSR[FRBp:FRBp+1] is			
F	∞	**QNaN**	**SNaN**
C(UIM:NSDb)	AuoB	AuoB	AuoB

Explanation:

C(UIM:NSDb)	Algebraic comparison. See the table below.
F	All finite numbers, including zeros.
AeqB	CR field BF and FPCC are set to 0b0010.
AgtB	CR field BF and FPCC are set to 0b0100.
AltB	CR field BF and FPCC are set to 0b1000.
AuoB	CR field BF and FPCC are set to 0b0001.

Relation of Value NSDb to Value UIM	Action for C(UIM:NSDb)
UIM $\neq$ 0 and UIM = NSDb	AeqB
UIM $\neq$ 0 and UIM < NSDb	AltB
UIM $\neq$ 0 and UIM > NSDb, or UIM = 0	AgtB

Figure 5.28: Actions: Test Significance

5.6.4 DFP Quantum Adjustment Instructions

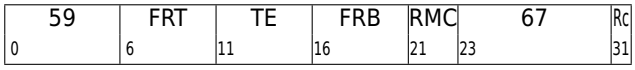
The *Quantum Adjustment* operations consist of the *Quantize*, *Quantize Immediate*, *Reround*, and *Round To FP Integer* operations.

The *Quantum Adjustment* instructions are Z23-form instructions and have an immediate RMC (Rounding-Mode-Control) field, which specifies the rounding mode used. For *Quantize*, *Quantize Immediate*, and *Reround*, the RMC field contains the primary encoding. For *Round to FP Integer*, the field contains either primary or secondary encoding, depending on the setting of a RMC-encoding-selection bit. See Section 5.5.2 “Rounding Mode Specification” on page 193 for the definition of RMC encoding.

All *Quantum Adjustment* instructions set the FI and FR status flags, and also set the FPRF field. The record bit is provided to each of these instructions. They return the target operand in a form with the ideal exponent.

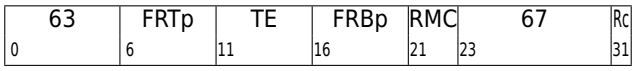
DFP Quantize Immediate Z23-form

dquai	TE,FRT,FRB,RMC	(Rc=0)
dquai.	TE,FRT,FRB,RMC	(Rc=1)



DFP Quantize Immediate Quad Z23-form

dquaiq	TE,FRTp,FRBp,RMC	(Rc=0)
dquaiq.	TE,FRTp,FRBp,RMC	(Rc=1)



The DFP operand in FRB[p] is converted and rounded to the form with the exponent specified by TE based on the rounding mode specified in the RMC field. TE is a 5-bit signed binary integer. The result of that form is placed in FRT[p]. The sign of the result is the same as the sign of the operand in FRB[p]. The ideal exponent is the exponent specified by TE.

When the value of the operand in FRB[p] is greater than $(10^{p}-1) \% 10^{TE}$, where p is the format precision, an invalid operation exception is recognized.

When the delivered result differs in value from the operand in FRB[p], an inexact exception is recognized. No underflow exception is recognized by this operation, regardless of the value of the operand in FRB[p].

FPRF is always set to the class and sign of the result, except for an enabled invalid-operation exception, in which case the field remains unchanged.

dquai[***q***][**.**] are treated as *Floating-Point* instructions in terms of resource availability.

Special Registers Altered:

Register	*Field(s)*	
FPSCR	FPRF FR FI FX XX VXSNAN VXCVI	
CR	CR1	(if Rc=1)

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 * 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 = −2) using Round to nearest, ties away from 0 (RMC = 2). The quantized and rounded result is placed in FRT.

```
dmul    f0,FRA,FRB
dquai   -2,FRT,f0,2
```

DFP Quantize Z23-form

dqua	FRT,FRA,FRB,RMC	(Rc=0)
dqua.	FRT,FRA,FRB,RMC	(Rc=1)

59	FRT	FRA	FRB	RMC	3	Rc
0	6	11	16	21	23	31

DFP Quantize Quad Z23-form

dquaq	FRTp,FRAp,FRBp,RMC	(Rc=0)
dquaq.	FRTp,FRAp,FRBp,RMC	(Rc=1)

63	FRTp	FRAp	FRBp	RMC	3	Rc
0	6	11	16	21	23	31

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 by RMC. 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 $(10^{p} - 1) \times 10^{Ea}$, 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 5.30 and Figure 5.31 summarize the actions. The tables do not include the setting of FPRF. FPRF is always set to the class and sign of the result, except for an enabled invalid-operation exception, in which case the field remains unchanged.

dqua[***q***][***.***] are treated as *Floating-Point* instructions in terms of resource availability.

Special Registers Altered:

Register	***Field(s)***	
FPSCR	FPRF FR FI FX XX VXSNAN VXCVI	
CR	CR1	(if Rc=1)

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 RMC. Figure 5.29 shows examples of these adjustments.

FRA	FRB	FRT when RMC=1	FRT when RMC=2
1 (1×10^{0})	9 (9×10^{0})	9 (9×10^{0})	9 (9×10^{0})
1.00 (100×10^{-2})	9. (9×10^{0})	9.00 (900×10^{-2})	9.00 (900×10^{-2})
1 (1×10^{0})	49.1234 (491234×10^{-4})	49 (49×10^{0})	49 (49×10^{0})
1.00 (100×10^{-2})	49.1234 (491234×10^{-4})	49.12 (4912×10^{-2})	49.12 (4912×10^{-2})
1 (1×10^{0})	49.9876 (499876×10^{-4})	49 (49×10^{0})	50 (50×10^{0})
1.00 (100×10^{-2})	49.9876 (499876×10^{-4})	49.98 (4998×10^{-2})	49.99 (4999×10^{-2})
0.01 (1×10^{-2})	49.9876 (499876×10^{-4})	49.98 (4998×10^{-2})	49.99 (4999×10^{-2})
1 (1×10^{0})	9999999999999999 ($9999999999999999 \times 10^{0}$)	9999999999999999 ($9999999999999999 \times 10^{0}$)	9999999999999999 ($9999999999999999 \times 10^{0}$)
1.0 (10×10^{-1})	9999999999999999 ($9999999999999999 \times 10^{0}$)	QNaN	QNaN

Figure 5.29: DFP Quantize examples

Operand a in FRA[p] is	Actions for Quantize when operand b in FRB[p] is				
	0	**Fn**	**∞**	**QNaN**	**SNaN**
0	*	*	VXCVI: T(dNaN)	P(b)	VXSNAN: U(b)
Fn	*	*	VXCVI: T(dNaN)	P(b)	VXSNAN: U(b)
∞	VXCVI: T(dNaN)	VXCVI: T(dNaN)	T(dINF)	P(b)	VXSNAN: U(b)
QNaN	P(a)	P(a)	P(a)	P(a)	VXSNAN: U(b)
SNaN	VXSNAN: U(a)	VXSNAN: U(a)	VXSNAN: U(a)	VXSNAN: U(a)	VXSNAN: U(a)

Explanation:

*	See next table.
dINF	Default infinity
dNaN	Default quiet NaN
Fn	Finite nonzero numbers (includes both subnormal and normal numbers)
P(x)	The QNaN of operand x is propagated and placed in FRT[p]
T(x)	The value x is placed in FRT[p]
U(x)	The SNaN of operand x is converted to the corresponding QNaN and placed in FRT[p].
VXCVI:	Floating-Point Invalid Operation (Invalid Conversion) exception occurs. The result is produced only when the exception is disabled. (See Section 5.5.10.1 “Invalid Operation Exception” on page 198.)
VXSNAN:	Floating-Point Invalid Operation (SNaN) exception occurs. The result is produced only when the exception is disabled. (See Section 5.5.10.1 “Invalid Operation Exception” on page 198.)

Figure 5.30: Actions (part 1) Quantize

	Actions for Quantize when operand b in FRB[p] is		
		0	**Fn**
Te $<$ Se	Vb $> (10^{p} - 1) \times 10^{Te}$	E(0)	VXCVI: T(dNaN)
	Vb $\leq (10^{p} - 1) \times 10^{Te}$	E(0)	L(b)
Te = Se		E(0)	W(b)
Te $>$ Se		E(0)	QR(b)

Explanation:

dNaN	Default quiet NaN
E(0)	The value of zero with the exponent value Te is placed in FRT[p].
L(x)	The operand x is converted to the form with the exponent value Te.
p	The precision of the format.
QR(x)	The operand x is rounded to the result of the form with the exponent value Te based on the specified rounding mode. The result of that form is placed in FRT[p].
Se	The exponent of the operand in FRB[p].
Te	The target exponent; FRA[p] for ***dqua***[***q***], or TE, a 5-bit signed binary integer for ***dquai***[***q***].
T(x)	The value x is placed in FRT[p].
Vb	The value of the operand in FRB[p].
W(x)	The value and the form of operand x is placed in FRT[p].
VXCVI:	Floating-Point Invalid Operation (Invalid Conversion) exception occurs. The result is produced only when the exception is disabled. (See Section 5.5.10.1 “Invalid Operation Exception” on page 198.)

Figure 5.31: Actions (part 2) Quantize

DFP Reround Z23-form

drrnd	FRT,FRA,FRB,RMC	(Rc=0)
drrnd.	FRT,FRA,FRB,RMC	(Rc=1)

59	FRT	FRA	FRB	RMC	35	Rc
0	6	11	16	21	23	31

DFP Reround Quad Z23-form

drrndq	FRTp,FRA,FRBp,RMC	(Rc=0)
drrndq.	FRTp,FRA,FRBp,RMC	(Rc=1)

63	FRTp	FRA	FRBp	RMC	35	Rc
0	6	11	16	21	23	31

Let k be the contents of bits 58:63 of FRA that specifies the reference significance.

When the DFP operand in FRB[p] is a finite number, and if the reference significance is zero, or if the reference significance is nonzero and the number of significant digits of the source operand is less than or equal to the reference significance, then the value and the form of the source operand is placed in FRT[p]. If the reference significance is nonzero and the number of significant digits of the source operand is greater than the reference significance, then the source operand is converted and rounded to the number of significant digits specified in the reference significance based on the rounding mode specified in the RMC field. The result of the form with the specified number of significant digits is placed in FRT[p]. The sign of the result is the same as the sign of the operand in FRB[p].

For this instruction, the number of significant digits of the value 0 is considered to be zero. The ideal exponent is the greater value of the exponent of the operand in FRB[p] and the referenced exponent. The referenced exponent is the resultant exponent if the operand in FRB[p] would have been converted and rounded to the number of significant digits specified in the reference significance based on the rounding mode specified by RMC.

If the exponent of the rounded result of the form that has the specified number of significant digits would be greater than X_{max}, an invalid operation exception (VXCVI) occurs. When the invalid-operation exception occurs, and if the exception is disabled, a default QNaN is returned. When an invalid-operation exception occurs, no inexact exception is recognized.

In the absence of an invalid-operation exception, if the result differs in value from the operand in FRB[p], an inexact exception is recognized.

This operation causes neither an overflow nor an underflow exception.

Figure 5.33 summarizes the actions for *Reround*. The table does not include the setting of FPRF. FPRF is always set to the class and sign of the result, except for an enabled invalid-operation exception, in which case the field remains unchanged.

***drrnd*[*q*][.]** are treated as *Floating-Point* instructions in terms of resource availability.

Special Registers Altered:

Register	*Field(s)*	
FPSCR	FPRF FR FI FX XX VXSNAN VXCVI	
CR	CR1	(if Rc=1)

Programming Note

DFP Reround can be used to adjust a DFP value (FRB[p]) to have no more than a specified number ($FRA[p]_{58:63}$) of significant digits. The result (FRT[p]) is right-justified leaving the specified number of digits and rounded as specified by RMC. If rounding increases the number of significant digits, the result is adjusted again (the significand is shifted right 1 digit and the exponent is incremented by 1). Figure 5.32 has example results from *DFP Reround* for 1, 2, and 10 significant digits.

Programming Note

DFP Reround is primarily used to round a DFP value to a specific number of digits before conversion to string format for printing or display. Another use for *DFP Reround* is to obtain the effective exponent of the most significant digit by specifying a reference significance of 1. The exponent can be extracted and used to compute the number of significant digits or to left-justify a value.

For example, the following sequence computes the number of significant digits and returns it as an integer. `FRB` is the DFP value for which we want the number of significant digits; `f13` contains the reference significance value `0x0000000000000001`; and `r1` is the stack pointer, with free space for doublewords at offsets -8 and -16. These doublewords are used to transfer the biased exponents from the FPRs to GPRs for integer computation. `R3` contains the result of `E(reround(1, FRA)) - E(FRA) + 1`, where `E(x)` represents the biased exponent of `x`.

```
dxex    f0,FRB
stfd    f0,-16(r1)
drrnd   f1,f13,FRB,1     # reround 1 digit toward 0
dxex    f1,f1
stfd    f1,-8(r1)
lfd     r11,-16(r1)
lfd     r3,-8(r1)
subf    r3,r11,r3
addi    r3,r3,1
```

Given the value 412.34 the result is `E(4 × 102) - E(41234 × 10-2) + 1 = (398+2) - (398-2) + 1 = 400 - 396 + 1 = 5`. Additional code is required to detect and handle special values like Subnormal, Infinity, and NAN.

$FRA_{58:63}$ (binary)	FRB	FRT when RMC=1	FRT when RMC=2
1	0.41234 (41234×10^{-5})	0.4 (4×10^{-1})	0.4 (4×10^{-1})
1	4.1234 (41234×10^{-4})	4 (4×10^{0})	4 (4×10^{0})
1	41.234 (41234×10^{-3})	4 (4×10^{1})	4 (4×10^{1})
1	412.34 (41234×10^{-2})	4 (4×10^{2})	4 (4×10^{2})
2	0.491234 (491234×10^{-6})	0.49 (49×10^{-2})	0.49 (49×10^{-2})
2	0.499876 (499876×10^{-6})	0.49 (49×10^{-2})	0.50 (50×10^{-2})
2	0.999876 (999876×10^{-6})	0.99 (99×10^{-2})	1.0 (10×10^{-1})
10	0.491234 (491234×10^{-6})	0.491234 (491234×10^{-6})	0.491234 (491234×10^{-6})
10	999.999 (999999×10^{-3})	999.999 (999999×10^{-3})	999.999 (999999×10^{-3})
10	9999999999999999 ($9999999999999999 \times 10^{0}$)	9.999999999E+14 (9999999999×10^{5})	1.000000000E+15 (1000000000×10^{6})

Figure 5.32: DFP Reround examples

Programming Note

DFP Reround combined with *DFP Quantize* can be used to left justify a value (as needed by the frexp function). 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.

```
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				
	0*	**Fn**	**∞**	**QNaN**	**SNaN**
k ≠ 0, k < m	-	RR(b) or VXCVI: T(dNaN)	T(dINF)	P(b)	VXSNAN: U(b)
k ≠ 0, k = m	-	W(b)	T(dINF)	P(b)	VXSNAN: U(b)
k ≠ 0 and k > m, or k = 0	W(b)	W(b)	T(dINF)	P(b)	VXSNAN: U(b)

Explanation:

*	The number of significant digits of the value 0 is considered to be zero for this instruction.
–	Not applicable.
dINF	Default infinity.
Fn	Finite nonzero numbers (includes both subnormal and normal numbers).
k	Reference significance, which specifies the number of significant digits in the target operand.
m	Number of significant digits in the operand in FRB[p].
P(x)	The QNaN of operand x is propagated and placed in FRT[p].
RR(x)	The value x is rounded to the form that has the specified number of significant digits. If RR(x) $\leq (10^k - 1) \times 10^{Xmax}$, then RR(x) is returned; otherwise an invalid-operation exception is recognized.
T(x)	The value x is placed in FRT[p].
U(x)	The SNaN of operand x is converted to the corresponding QNaN and placed in FRT[p].
VXCVI	Floating-Point Invalid Operation (Invalid Conversion) exception occurs. The result is produced only when the exception is disabled. (See Section 5.5.10.1 “Invalid Operation Exception” on page 198.)
VXSNAN:	Floating-Point Invalid Operation (SNaN) exception occurs. The result is produced only when the exception is disabled. (See Section 5.5.10.1 “Invalid Operation Exception” on page 198.)
W(x)	The value and the form of x is placed in FRT[p].

Figure 5.33: Actions: Reround

DFP Round To FP Integer With Inexact Z23-form

drintx	R,FRT,FRB,RMC	(Rc=0)
drintx.	R,FRT,FRB,RMC	(Rc=1)

59	FRT	///	R	FRB	RMC	99	Rc
0	6	11	15	16	21	23	31

DFP Round To FP Integer With Inexact Quad Z23-form

drintxq	R,FRTp,FRBp,RMC	(Rc=0)
drintxq.	R,FRTp,FRBp,RMC	(Rc=1)

63	FRTp	///	R	FRBp	RMC	99	Rc
0	6	11	15	16	21	23	31

The DFP operand in FRB[p] is rounded to a floating-point integer and placed into FRT[p]. The sign of the result is the same as the sign of the operand in FRB[p]. The ideal exponent is the larger value of zero and the exponent of the operand in FRB[p].

The rounding mode used is specified by RMC. When the RMC-encoding-selection (R) bit is zero, RMC field contains the primary encoding; when the bit is one, the field contains the secondary encoding.

In addition to coercion of the converted value to fit the target format, the special rounding used by *Round To FP Integer* also coerces the target exponent to the ideal exponent.

When the operand in FRB[p] is a finite number and the exponent is less than zero, the operand is rounded to the result with an exponent of zero. When the exponent is greater than or equal to zero, the result is set to the numerical value and the form of the operand in FRB[p].

When the result differs in value from the operand in FRB[p], an inexact exception is recognized. No underflow exception is recognized by this operation, regardless of the value of the operand in FRB[p].

Figure 5.34 summarizes the actions for *Round To FP Integer With Inexact*. The table does not include the setting of FPRF. FPRF is always set to the class and sign of the result, except for an enabled invalid-operation, in which case the field remains unchanged.

drintx[***q***][**.**] are treated as *Floating-Point* instructions in terms of resource availability.

Special Registers Altered:

Register	***Field(s)***	
FPSCR	FPRF FR FI FX XX VXSNAN	
CR	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 DRN (R=0, RMC=11). The specification for rint requires the inexact exception be raised if detected.

Operand b in FRB is	Is n not precise (n ≠ b)	Inv.-Op. Exception Enabled	Inexact Exception Enabled	Is n Incremented (|n| > |b|)	Actions*
−∞	No[1]	–	–	–	T(-dINF), 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
+∞	No[1]	–	–	–	T(+dINF), 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

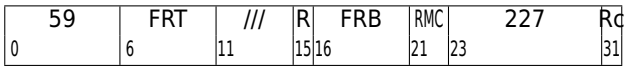
Explanation:

- * 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.
- 1 This condition is true by virtue of the state of some condition to the left of this column.
- dINF Default infinity.
- F All finite numbers, including zeros.
- FI Floating-Point Fraction Inexact status bit.
- FR Floating-Point Fraction Rounded status bit.
- n The value derived when the source operand, b, is rounded to an integer using the special rounding for *Round To FP Integer*.
- P(x) The QNaN of operand x is propagated and placed in FRT[p].
- T(x) The value x is placed in FRT[p].
- TV The system floating-point enabled exception error handler is invoked for the invalid operation exception if FE0 and FE1 are set to any mode other than the ignore-exception mode.
- TX The system floating-point enabled exception error handler is invoked for the inexact exception if FE0 and FE1 are set to any mode other than the ignore-exception mode.
- U(x) The SNaN of operand x is converted to the corresponding QNaN and placed in FRT[p].
- W(x) The value x in the form of zero exponent or the source exponent is placed in FRT[p].
- XX Floating-Point Inexact exception status bit.

Figure 5.34: Actions: Round to FP Integer With Inexact

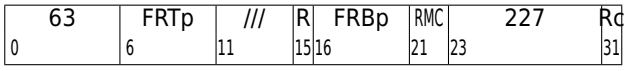
DFP Round To FP Integer Without Inexact Z23-form

drintn	R,FRT,FRB,RMC	(Rc=0)
drintn.	R,FRT,FRB,RMC	(Rc=1)



DFP Round To FP Integer Without Inexact Quad Z23-form

drintnq	R,FRTp,FRBp,RMC	(Rc=0)
drintnq.	R,FRTp,FRBp,RMC	(Rc=1)



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 5.35 summarizes the actions for *Round To FP Integer Without Inexact*. The table does not include the setting of FPRF. FPRF is always set to the class and sign of the result, except for an enabled invalid-operation, in which case the field remains unchanged.

drintn[***q***][**.**] are treated as *Floating-Point* instructions in terms of resource availability.

Special Registers Altered:

Register	*Field(s)*	
FPSCR	FPRF FX VXSNAN	
FPSCR	FR (set to 0)	
FPSCR	FI (set to 0)	
CR	CR1	(if Rc=1)

Programming Note

The *DFP Round To FP Integer Without Inexact* and *DFP Round To FP Integer Without Inexact Quad* instructions can be used to implement decimal equivalents of several C99 rounding functions by specifying the appropriate R and RMC field values.

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.

Operand b in FRB is	Inv.-Op. Exception Enabled	Actions*
$-\infty$	-	T(-dINF), FI←0, FR←0
F	-	W(n), FI←0, FR←0
$+\infty$	-	T(+dINF), FI←0, FR←0
QNaN	-	P(b), FI←0, FR←0
SNaN	No	U(b), FI←0, FR←0, VXSNAN←1
SNaN	Yes	VXSNAN←1, TV

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.
- dINF Default infinity.
- F All finite numbers, including zeros.
- FI Floating-Point Fraction Inexact status bit.
- FR Floating-Point Fraction Rounded status bit.
- n The value derived when the source operand, b, is rounded to an integer using the special rounding for Round To FP Integer.
- P(x) The QNaN of operand x is propagated and placed in FRT[p].
- T(x) The value x is placed in FRT[p].
- TV The system floating-point enabled exception error handler is invoked for the invalid-operation exception if FE0 and FE1 are set to any mode other than the ignore-exception mode.
- U(x) The SNaN of operand x is converted to the corresponding QNaN and placed in FRT[p].
- W(x) The value x in the form of zero exponent or the source exponent is placed in FRT[p].

Figure 5.35: 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 5.36 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.

Instruction	Actions when operand b in FRB[p] is			
	F	∞	QNaN	SNaN
Convert To DFP Long	T(b)[1]	P(b)[2,4]	P(b)[2,4]	P(b)[3,4]
Convert To DFP Extended	T(b)[1]	T(dINF)	P(b)[2,4]	VXSNAN: U(b)[2,4]
Round To DFP Short	R(b)[1]	P(b)[2,5]	P(b)[2,5]	P(b)[3,5]
Round To DFP Long	R(b)[1]	T(dINF)	P(b)[2,5]	VXSNAN: U(b)[2,5]

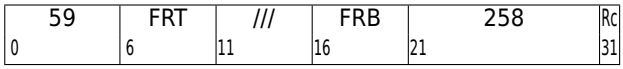
Explanation:

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.
dINF	Default infinity.
F	All finite numbers, including zeros.
P(x)	The special symbol in operand x is propagated into FRT[p].
R(x)	The value x is rounded to the target-format precision; see Section 5.5.11
T(x)	The value x is placed in FRT[p].
U(x)	The SNaN of operand x is converted to the corresponding QNaN.
VXSNAN	Floating-Point Invalid Operation (SNaN) exception occurs. The result is produced only when the exception is disabled. (See Section 5.5.10.1 “Invalid Operation Exception” on page 198.)

Figure 5.36: Actions: Data-Format Conversion Instructions

DFP Convert To DFP Long X-form

dctdp	FRT,FRB	(Rc=0)
dctdp.	FRT,FRB	(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.

dctdp[**.**] are treated as *Floating-Point* instructions in terms of resource availability.

Special Registers Altered:

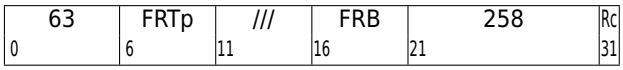
Register	*Field(s)*	
FPSCR	FPRF	
FPSCR	FR (undefined)	
FPSCR	FI (undefined)	
CR	CR1	(if Rc=1)

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 X-form

dctqpq	FRTp,FRB	(Rc=0)
dctqpq.	FRTp,FRB	(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.

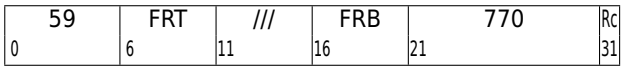
dctqpq[**.**] are treated as *Floating-Point* instructions in terms of resource availability.

Special Registers Altered:

Register	*Field(s)*	
FPSCR	FPRF FX VXSNAN	
FPSCR	FR (set to 0)	
FPSCR	FI (set to 0)	
CR	CR1	(if Rc=1)

DFP Round To DFP Short X-form

drsp	FRT,FRB	(Rc=0)
drsp.	FRT,FRB	(Rc=1)



The DFP long operand in FRB is converted and rounded to DFP short format. The DFP short value is extended on the left with zeros to form a 64-bit entity and placed into FRT. The sign of the result is the same as the sign of the source operand. The ideal exponent is the exponent of the source operand.

If the operand in FRB is an SNaN, it is converted to an SNaN in DFP short format and does not cause an invalid-operation exception.

Normally, the result is in the format and length of the target. However, when an overflow or underflow exception occurs and if the exception is enabled, the operation is completed by producing a wrapped rounded result in the same format and length as the source but rounded to the target-format precision.

drsp[**.**] are treated as *Floating-Point* instructions in terms of resource availability.

Special Registers Altered:

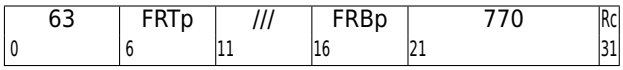
Register	*Field(s)*	
FPSCR	FPRF FR FI FX OX UX XX	
CR	CR1	(if Rc=1)

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.

DFP Round To DFP Long X-form

drdpq	FRTp,FRBp	(Rc=0)
drdpq.	FRTp,FRBp	(Rc=1)



The DFP extended operand in FRBp is converted and rounded to DFP long format. The result concatenated with 64 0s is placed in FRTp. The sign of the result is the same as the sign of the source operand. The ideal exponent is the exponent of the operand in FRBp.

If the operand in FRBp is an SNaN, an invalid-operation exception is recognized. If the exception is disabled, the SNaN is converted to the corresponding QNaN in DFP long format.

Normally, the result is in the format and length of the target. However, when an overflow or underflow exception occurs and if the exception is enabled, the operation is completed by producing a wrapped rounded result in the same format and length as the source but rounded to the target-format precision.

drdpq[**.**] are treated as *Floating-Point* instructions in terms of resource availability.

Special Registers Altered:

Register	*Field(s)*	
FPSCR	FPRF FR FI FX OX UX XX VXSNAN	
CR	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.

The data-type conversion instructions consist of *Convert From Fixed* and *Convert To Fixed*.

DFP Convert From Fixed X-form

dcffix	FRT,FRB	(Rc=0)
dcffix.	FRT,FRB	(Rc=1)

59	FRT	///	FRB	802	Rc
0	6	11	16	21	31

The 64-bit signed binary integer in FRB is converted and rounded to a DFP Long value and placed into FRT. The sign of the result is the same as the sign of the source operand. The ideal exponent is zero.

If the source operand is a zero, then a plus zero with a zero exponent is returned.

FPRF is set to the class and sign of the result.

***dcffix*[*.*]** are treated as *Floating-Point* instructions in terms of resource availability.

Special Registers Altered:

Register	*Field(s)*	
FPSCR	FPRF FR FI FX XX	
CR	CR1	(if Rc=1)

DFP Convert From Fixed Quad X-form

dcffixq	FRTp,FRB	(Rc=0)
dcffixq.	FRTp,FRB	(Rc=1)

63	FRTp	///	FRB	802	Rc
0	6	11	16	21	31

The 64-bit signed binary integer in FRB is converted and rounded to a DFP Extended value and placed into FRTp. The sign of the result is the same as the sign of the source operand. The ideal exponent is zero.

If the source operand is a zero, then a plus zero with a zero exponent is returned.

FPRF is set to the class and sign of the result.

***dcffixq*[*.*]** are treated as *Floating-Point* instructions in terms of resource availability.

Special Registers Altered:

Register	*Field(s)*	
FPSCR	FPRF	
FPSCR	FR (undefined)	
FPSCR	FI (undefined)	
CR	CR1	(if Rc=1)

DFP Convert From Fixed Quadword Quad X-form

dcffixqq	FRTp,VRB

63	FRTp	0	VRB	994	/
0	6	11	16	21	31

The 128-bit signed binary integer in VRB is converted and rounded to a DFP Extended value and placed into FRTp. The sign of the result is the same as the sign of the source operand. The ideal exponent is zero.

If the source operand is a zero, then a plus zero with a zero exponent is returned.

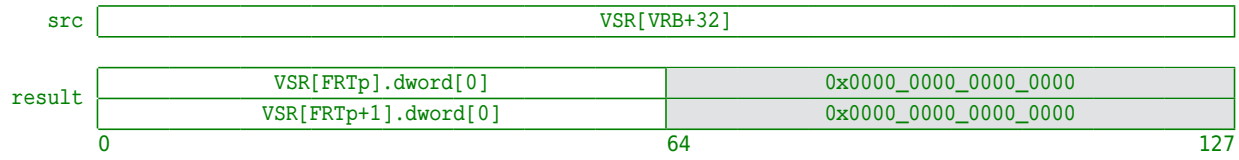
FPRF is set to the class and sign of the result.

dcffixqq is treated as a *Floating-Point* and a *Vector* instruction in terms of resource availability.

Special Registers Altered:

Register	*Field(s)*
FPSCR	FPRF FR FI FX XX

VSR Data Layout for dcffixqq



DFP Convert To Fixed X-form

dctfix FRT,FRB (Rc=0)
dctfix. FRT,FRB (Rc=1)

59	FRT	///	FRB	290	Rc
0	6	11	16	21	31

DFP Convert To Fixed Quad X-form

dctfixq FRT,FRBp (Rc=0)
dctfixq. FRT,FRBp (Rc=1)

63	FRT	///	FRBp	290	Rc
0	6	11	16	21	31

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 5.37 summarizes the actions for *Convert To Fixed*.

dctfix[***q***][**.**] are treated as *Floating-Point* instructions in terms of resource availability.

Special Registers Altered:

Register	*Field(s)*	
FPSCR	FPRF (undefined)	
FPSCR	FX XX VXSNAN VXCVI	
CR	CR1	(if Rc=1)

Programming Note

It is recommended that software pre-round the operand to a floating-point integral using ***drintx***[***q***] or ***drintn***[***q***] if a rounding mode other than the current rounding mode specified by DRN is needed. Saving, modifying and restoring the FPSCR just to temporarily change the rounding mode is less efficient than just employing drintx[p] or drint[p] which override the current rounding mode using an immediate control field.

For example if the desired function rounding is Round to Nearest, Ties away from 0 but the default rounding (from DRN) is Round to Nearest, Ties to Even then following is preferred.

```
drintn   0,f1,f1,2
dctfix   f1,f1
```

DFP Convert To Fixed Quadword Quad X-form

dctfixqq VRT,FRBp

63	VRT	1	FRBp	994	/
0	6	11	16	21	31

The DFP operand in FRBp is rounded to an integer value and is placed into VRT in the 128-bit signed binary integer format. The sign of the result is the same as the sign of the source operand, except when the source operand is a NaN or a zero.

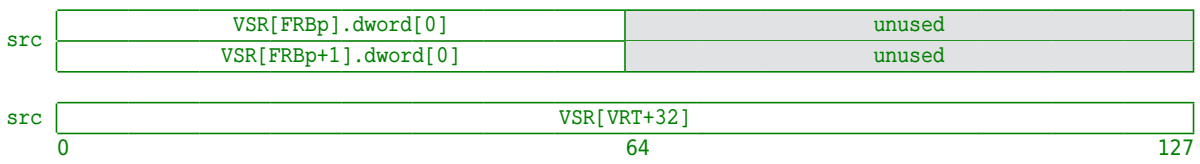
Figure 5.37 summarizes the actions for *Convert To Fixed*.

dctfixqq is treated as a Floating-Point and a Vector instruction in terms of resource availability.

Special Registers Altered:

Register	*Field(s)*
FPSCR	FPRF (undefined)
FPSCR	FR FI FX VXSNAN VXCVI XX

VSR Data Layout for dctfixqq



Operand b in FRB[p] is	q is	Is n not precise (n ≠ b)	Inv.-Op. Except. Enabled	Inexact Except. Enabled	Is n Incremented (|n| > |b|)	Actions *
$-\infty \le b < MN$	$< MN$	–	No	–	–	T(MN), FI←0, FR←0, VXCVI←1
$-\infty \le b < MN$	$< MN$	–	Yes	–	–	VXCVI←1, TV
$-\infty < b < MN$	$= MN$	–	–	No	–	T(MN), FI ← 1, FR←0, XX←1
$-\infty < b < MN$	$= MN$	–	–	Yes	–	T(MN), FI ← 1, FR←0, XX←1, TX
$MN \le b < 0$	–	No	–	–	–	T(n), FI←0, FR←0
$MN \le b < 0$	–	Yes	–	No	No	T(n), FI←1, FR←0, XX←1
$MN \le b < 0$	–	Yes	–	No	Yes	T(n), FI←1, FR←1, XX←1
$MN \le b < 0$	–	Yes	–	Yes	No	T(n), FI←1, FR←0, XX←1, TX
$MN \le b < 0$	–	Yes	–	Yes	Yes	T(n), FI←1, FR←1, XX←1, TX
± 0	–	No	–	–	–	T(0), FI←0, FR←0
$0 < b \le MP$	–	No	–	–	–	T(n), FI←0, FR←0
$0 < b \le MP$	–	Yes	–	No	No	T(n), FI←1, FR←0, XX←1
$0 < b \le MP$	–	Yes	–	No	Yes	T(n), FI←1, FR←1, XX←1
$0 < b \le MP$	–	Yes	–	Yes	No	T(n), FI←1, FR←0, XX←1, TX
$0 < b \le MP$	–	Yes	–	Yes	Yes	T(n), FI←1, FR←1, XX←1, TX
$MP < b < +\infty$	$= MP$	–	–	No	–	T(MP), FI←1, FR←0, XX←1
$MP < b < +\infty$	$= MP$	–	–	Yes	–	T(MP), FI←1, FR←0, XX←1, TX
$MP < b \le +\infty$	$> MP$	–	No	–	–	T(MP), FI←0, FR←0, VXCVI←1
$MP < b \le +\infty$	$> MP$	–	Yes	–	–	VXCVI←1, TV
QNaN	–	–	No	–	–	T(MN), FI ← 0, FR←0, VXCVI←1
QNaN	–	–	Yes	–	–	VXCVI←1, TV
SNaN	–	–	No	–	–	T(MN), FI←0, FR←0, VXCVI←1, VXSNAN←1
SNaN	–	–	Yes	–	–	VXCVI←1, VXSNAN←1, TV

Explanation:

*	Setting of XX, VXCVI, and VXSNAN is part of the corresponding exception actions. Also, when an invalid-operation exception occurs, setting of FI and FR bits is part of the exception actions. (See the sections, “Inexact Exception” and “Invalid Operation Exception” for more details.)
–	The actions do not depend on this condition.
FI	Floating-Point Fraction Inexact status bit.
FR	Floating-Point Fraction Rounded status bit.
MN	Maximum negative number representable by the 64-bit binary integer format
MP	Maximum positive number representable by the 64-bit binary integer format.
n	The value q converted to a fixed-point result.
q	The value derived when the source value b is rounded to an integer using the specified rounding mode
T(x)	The value x is placed in FRT[p].
TV	The system floating-point enabled exception error handler is invoked for the invalid-operation exception if FE0 and FE1 are set to any mode other than the ignore-exception mode.
TX	The system floating-point enabled exception error handler is invoked for the inexact exception if FE0 and FE1 are set to any mode other than the ignore-exception mode.
VXCVI	Floating-Point Invalid Operation (Invalid Conversion) exception status bit.
VXSNAN	Floating-Point Invalid Operation (SNaN) exception status bit.
XX	Floating-Point Inexact exception status bit.

Figure 5.37: Actions: Convert To Fixed

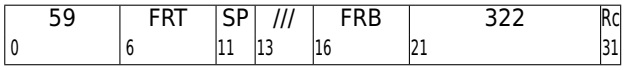
5.6.6 DFP Format Instructions

The DFP format instructions are used to compose or decompose a DFP operand. A source operand of SNaN does not cause an invalid-operation exception. All format instructions employ the record bit (Rc).

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*.

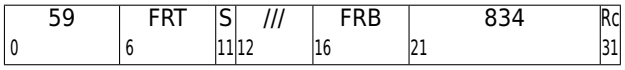
DFP Decode DPD To BCD X-form

ddedpd	SP,FRT,FRB	(Rc=0)
ddedpd.	SP,FRT,FRB	(Rc=1)



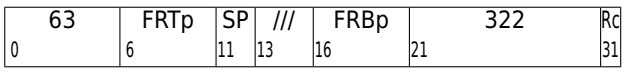
DFP Encode BCD To DPD X-form

denbcd	S,FRT,FRB	(Rc=0)
denbcd.	S,FRT,FRB	(Rc=1)



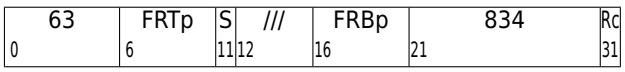
DFP Decode DPD To BCD Quad X-form

ddedpdq	SP,FRTp,FRBp	(Rc=0)
ddedpdq.	SP,FRTp,FRBp	(Rc=1)



DFP Encode BCD To DPD Quad X-form

denbcdq	S,FRTp,FRBp	(Rc=0)
denbcdq.	S,FRTp,FRBp	(Rc=1)



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.

SP_0 = 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]`.

SP_0 = 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, SP_1 indicates which preferred plus sign encoding is used. If SP_1 = 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).

ddedpd[***q***][***.***] are treated as *Floating-Point* instructions in terms of resource availability.

Special Registers Altered:

Register	*Field(s)*	
CR	CR1	(if Rc=1)

The signed or unsigned BCD operand, depending on the S field, in `FRB[p]` is converted to a DFP number. The ideal exponent is zero.

S = 0 (unsigned BCD operand)

The unsigned BCD operand in `FRB[p]` is converted to a positive DFP number of the same magnitude and the result is placed into `FRT[p]`.

S = 1 (signed BCD operand)

The signed BCD operand in `FRB[p]` is converted to the corresponding DFP number and the result is placed into `FRT[p]`.

If an invalid BCD digit or sign code is detected in the source operand, an invalid-operation exception (`VXCVI`) occurs.

`FPRF` is set to the class and sign of the result, except for Invalid Operation Exception when `VE=1`.

denbcd[***q***][***.***] are treated as *Floating-Point* instructions in terms of resource availability.

Special Registers Altered:

Register	*Field(s)*	
FPSCR	FPRF FX VXCVI	
FPSCR	FR (set to 0)	
FPSCR	FI (set to 0)	
CR	CR1	(if Rc=1)

DFP Extract Biased Exponent X-form

dxex	FRT,FRB	(Rc=0)
dxex.	FRT,FRB	(Rc=1)

59	FRT	///	FRB	354	Rc
0	6	11	16	21	31

DFP Extract Biased Exponent Quad X-form

dxexq	FRT,FRBp	(Rc=0)
dxexq.	FRT,FRBp	(Rc=1)

63	FRT	///	FRBp	354	Rc
0	6	11	16	21	31

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	Result
Finite Number	biased exponent value
Infinity	-1
QNaN	-2
SNaN	-3

dxex[***q***][**.**] are treated as *Floating-Point* instructions in terms of resource availability.

Special Registers Altered:

Register	*Field(s)*	
CR	CR1	(if Rc=1)

Programming Note

The exponent bias value is 101 for DFP Short, 398 for DFP Long, and 6176 for DFP Extended.

DFP Insert Biased Exponent X-form

diex	FRT,FRA,FRB	(Rc=0)
diex.	FRT,FRA,FRB	(Rc=1)

59	FRT	FRA	FRB	866	Rc
0	6	11	16	21	31

DFP Insert Biased Exponent Quad X-form

diexq	FRTp,FRA,FRBp	
diexq.	FRTp,FRA,FRBp	(Rc=1)

63	FRTp	FRA	FRBp	866	Rc
0	6	11	16	21	31

Let *a* be the value of the 64-bit signed binary integer in FRA.

a	**Result**
$a > MBE$[1]	QNaN
$0 \le a \le MBE$	Finite number with biased exponent a
$a = -1$	Infinity
$a = -2$	QNaN
$a = -3$	SNaN
$a < -3$	QNaN

[1] Maximum biased exponent for the target format

When $0 \le a \le MBE$, 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.

diex[***q***][**.**] are treated as *Floating-Point* instructions in terms of resource availability.

Special Registers Altered:

Register	*Field(s)*	
CR	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	Z, Rb
∞	I, Rb	I, Rb	I, Rb	I, Rb
QNaN	Q, Rb	Q, Rb	Q, Rb	Q, Rb
SNaN	S, Rb	S, Rb	S, Rb	S, 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 5.38: Actions: Insert Biased Exponent

DFP Shift Significand Left Immediate Z22-form

dscli	FRT,FRA,SH	(Rc=0)
dscli.	FRT,FRA,SH	(Rc=1)

59	FRT	FRA	SH	66	Rc
0	6	11	16	22	31

DFP Shift Significand Left Immediate Quad Z22-form

dscliq	FRTp,FRAp,SH	(Rc=0)
dscliq.	FRTp,FRAp,SH	(Rc=1)

63	FRTp	FRAp	SH	66	Rc
0	6	11	16	22	31

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.

dscli[***q***][***.***] are treated as *Floating-Point* instructions in terms of resource availability.

Special Registers Altered:

Register	*Field(s)*	
CR	CR1	(if Rc=1)

DFP Shift Significand Right Immediate Z22-form

dscri	FRT,FRA,SH	(Rc=0)
dscri.	FRT,FRA,SH	(Rc=1)

59	FRT	FRA	SH	98	Rc
0	6	11	16	22	31

DFP Shift Significand Right Immediate Quad Z22-form

dscriq	FRTp,FRAp,SH	(Rc=0)
dscriq.	FRTp,FRAp,SH	(Rc=1)

63	FRTp	FRAp	SH	98	Rc
0	6	11	16	22	31

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.

dscri[***q***][***.***] are treated as *Floating-Point* instructions in terms of resource availability.

Special Registers Altered:

Register	*Field(s)*	
CR	CR1	(if Rc=1)

5.6.7 DFP Instruction Summary

Table 5.1: Decimal Floating-Point Instructions Summary

Mnemonic	Full Name	FORM	Operands	SNaN		Encoding	FPRF		FP Exception					FR/FI	IE	Rc
				Vs	G		C	FPCC	V	Z	O	U	X			
dadd	DFP Add	X	FRT, FRA, FRB	Y	N	RE	Y	Y	V		O	U	X	Y	Y	Y
daddq	DFP Add Quad	X	FRTp, FRAp, FRBp	Y	N	RE	Y	Y	V		O	U	X	Y	Y	Y
dsub	DFP Subtract	X	FRT, FRA, FRB	Y	N	RE	Y	Y	V		O	U	X	Y	Y	Y
dsubq	DFP Subtract Quad	X	FRTp, FRAp, FRBp	Y	N	RE	Y	Y	V		O	U	X	Y	Y	Y
dmul	DFP Multiply	X	FRT, FRA, FRB	Y	N	RE	Y	Y	V		O	U	X	Y	Y	Y
dmulq	DFP Multiply Quad	X	FRTp, FRAp, FRBp	Y	N	RE	Y	Y	V		O	U	X	Y	Y	Y
ddiv	DFP Divide	X	FRT, FRA, FRB	Y	N	RE	Y	Y	V	Z	O	U	X	Y	Y	Y
ddivq	DFP Divide Quad	X	FRTp, FRAp, FRBp	Y	N	RE	Y	Y	V	Z	O	U	X	Y	Y	Y
dcmpo	DFP Compare Ordered	X	BF, FRA, FRB	Y	–	-	N	Y	V					–	–	N
dcmpoq	DFP Compare Ordered Quad	X	BF, FRAp, FRBp	Y	–	–	N	Y	V					–	–	N
dcmpu	DFP Compare Unordered	X	BF, FRA, FRB	Y	–	–	N	Y	V					–	–	N
dcmpuq	DFP Compare Unordered Quad	X	BF, FRAp, FRBp	Y	–	–	N	Y	V					–	–	N
dtstdc	DFP Test Data Class	Z22	BF, FRA, DCM	N	–	–	N	Y[1]						–	–	N
dtstdcq	DFP Test Data Class Quad	Z22	BF, FRAp, DCM	N	–	–	N	Y[1]						–	–	N
dtstdg	DFP Test Data Group	Z22	BF, FRA,DGM	N	–	–	N	Y[1]						–	–	N
dtstdgq	DFP Test Data Group Quad	Z22	BF, FRAp, DGM	N	–	–	N	Y[1]						–	–	N
dtstex	DFP Test Exponent	X	BF, FRA, FRB	N	–	–	N	Y						–	–	N
dtstexq	DFP Test Exponent Quad	X	BF, FRAp, FRBp	N	–	–	N	Y						–	–	N
dtstsf	DFP Test Significance	X	BF, FRA(FIX), FRB	N	–	–	N	Y						–	–	N
dtstsfq	DFP Test Significance Quad	X	BF, FRA(FIX), FRBp	N	–	–	N	Y						–	–	N
dtstsfi	DFP Test Significance Immediate	X	BF, UIM, FRB	N	–	–	N	Y						–	–	N
dtstsfiq	DFP Test Significance Immediate Quad	X	BF, UIM, FRBp	N	–	–	N	Y						–	–	N
dquai	DFP Quantize Immediate	Z23	TE, FRT, FRB, RMC	Y	N	RE	Y	Y	V				X	Y	Y	Y
dquaiq	DFP Quantize Immediate Quad	Z23	TE, FRTp, FRBp, RMC	Y	N	RE	Y	Y	V				X	Y	Y	Y
dqua	DFP Quantize	Z23	FRT,FRA,FRB,RMC	Y	N	RE	Y	Y	V				X	Y	Y	Y
dquaq	DFP Quantize Quad	Z23	FRTp,FRAp,FRBp, RMC	Y	N	RE	Y	Y	V				X	Y	Y	Y
drrnd	DFP Reround	Z23	FRT,FRA(FIX),FRB,RMC	Y	N	RE	Y	Y	V				X	Y	Y	Y
drrndq	DFP Reround Quad	Z23	FRTp, FRA(FIX), FRBp, RMC	Y	N	RE	Y	Y	V				X	Y	Y	Y
drintx	DFP Round To FP Integer With Inexact	Z23	R,FRT, FRB,RMC	Y	N	RE	Y	Y	V				X	Y	Y	Y
drintxq	DFP Round To FP Integer With Inexact Quad	Z23	R,FRTp,FRBp,RMC	Y	N	RE	Y	Y	V				X	Y	Y	Y
drintn	DFP Round To FP Integer Without Inexact	Z23	R,FRT, FRB,RMC	Y	N	RE	Y	Y	V					Y[#]	Y	Y
drintnq	DFP Round To FP Integer Without Inexact Quad	Z23	R,FRTp, FRBp,RMC	Y	N	RE	Y	Y	V					Y[#]	Y	Y
dctdp	DFP Convert To DFP Long	X	FRT, FRB (DFP Short)	N	Y	RE	Y	Y[2]						U	Y	Y
dctqpq	DFP Convert To DFP Extended	X	FRTp, FRB	Y	N	RE	Y	Y	V					Y[#]	Y	Y
drsp	DFP Round To DFP Short	X	FRT (DFP Short), FRB	N	Y	RE	Y	Y[2]			O	U	X	Y	Y	Y
drdpq	DFP Round To DFP Long	X	FRTp, FRBp	Y	N	RE	Y	Y	V		O	U	X	Y	Y	Y
dcffixq	DFP Convert From Fixed Quad	X	FRTp, FRB (FIX)	–	N	RE	Y	Y						U	Y	Y

Table 5.1: Decimal Floating-Point Instructions Summary (continued)

Mnemonic	Full Name	FORM	Operands	SNaN		Encoding	FPRF		FP Exception					FR/FI	IE	Rc
				Vs	G		C	FPCC	V	Z	O	U	X			
dctfix	DFP Convert To Fixed	X	FRT (FIX), FRB	Y	N	-	U	U	V				X	Y	–	Y
dctfixq	DFP Convert To Fixed Quad	X	FRT (FIX), FRBp	Y	N	–	U	U	V				X	Y	–	Y
dcffixqq	DFP Convert From Fixed Quad-word Quad	x	FRTp,VRB (FIX)	–	N	–	Y	Y						U	Y	Y
dctfixqq	DFP Convert To Fixed Quad-word Quad	x	VRT (FIX),FRBp	Y	N	–	U	U	V				X	Y	–	Y
ddedpd	DFP Decode DPD To BCD	X	SP, FRT(BCD), FRB	N	–	–	N	N						–	–	Y
ddedpdq	DFP Decode DPD To BCD Quad	X	SP, FRTp(BCD), FRBp	N	–	–	N	N						–	–	Y
denbcd	DFP Encode BCD To DPD	X	S, FRT, FRB (BCD)	–	N	RE	Y	Y	V					Y[#]	Y	Y
denbcdq	DFP Encode BCD To DPD Quad	X	S, FRTp, FRBp (BCD)	–	N	RE	Y	Y	V					Y[#]	Y	Y
dxex	DFP Extract Biased Exponent	X	FRT (FIX), FRB	N	N	–	N	N						–	–	Y
dxexq	DFP Extract Biased Exponent Quad	X	FRT (FIX), FRBp	N	N	–	N	N						–	–	Y
diex	DFP Insert Biased Exponent	X	FRT, FRA(FIX), FRB	N	Y	RE	N	N						–	Y	Y
diexq	DFP Insert Biased Exponent Quad	X	FRTp, FRA(FIX), FRBp	N	Y	RE	N	N						–	Y	Y
dscli	DFP Shift Significand Left Immediate	Z22	FRT,FRA,SH	N	Y	RE	N	N						–	–	Y
dscliq	DFP Shift Significand Left Immediate Quad	Z22	FRTp,FRAp,SH	N	Y	RE	N	N						–	–	Y
dscri	DFP Shift Significand Right Immediate	Z22	FRT,FRA,SH	N	Y	RE	N	N						–	–	Y
dscriq	DFP Shift Significand Right Immediate Quad	Z22	FRTp,FRAp,SH	N	Y	RE	N	N						–	–	Y

Explanation:

[#]	FI and FR are set to zeros for these instructions.
–	Not applicable.
[1]	A unique definition of the $FPSCR_{FPCC}$ field is provided for the instruction.
[2]	These are the only instructions that may generate an SNaN and also set the $FPSC_{FPRF}$ field. Since the BFP FPRF field does not include a code for SNaN, these instructions cause the need for redefining the FPRF field for DFP.
DCM	A 6-bit immediate operand specifying the data-class mask.
DGM	A 6-bit immediate operand specifying the data-group mask.
G	An SNaN can be generated as the target operand.
IE	An ideal exponent is defined for the instruction.
FI	Setting of the $FPSCR_{FI}$ flag.
FR	Setting of the $FPSCR_{FR}$ flag.
N	No.
O	An overflow exception may be recognized.
Rc	The record bit, Rc, is provided to record $FPSCR_{32:35}$ in CR field 1.
RE	The trailing significand field is reencoded using preferred DPD encodings.The preferred DPD encoding are also used for propagated NaNs, or converted NaNs and infinities.
RMC	A 2-bit immediate operand specifying the rounding-mode control.
S	An one-bit immediate operand specifying if the operation is signed or unsigned.
SP	A two-bit immediate operand: one bit specifies if the operation is signed or unsigned and, for signed operations, another bit specifies which preferred plus sign code is generated.
U	An underflow exception may be recognized.

Explanation:

V	An invalid-operation exception may be recognized.
Vs	An input operand of SNaN causes an invalid-operation exception.
X	An inexact exception may be recognized.
Y	Yes.
U	Undefined
Z	A zero-divide exception may be recognized.

Chapter 6. Vector Facility

6.1 Vector Facility Overview

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

6.2 Chapter Conventions

6.2.1 Description of Instruction Operation

The following notation, in addition to that described in Section 1.3.2, is used in this chapter.

[]

x.bit[y:z]
Return the contents of bits y:z of x.

[]

x.nibble[y:z]
Return the contents of the 4-bit nibble elements y:z of x.

[]

x.byte[y:z]
Return the contents of 8-bit byte elements y:z of x.

[]

x.hword[y:z]
Return the contents of 16-bit halfword elements y:z of x.

[]

x.word[y:z]
Return the contents of 32-bit word element y:z of x.

[]

x.dword[y:z]
Return the contents of 64-bit 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.

+ Addition.

- Subtraction.

$\times$ Multiplication.

$\neg$ One's complement.

=, <, <=, >, >=
Equal, less than, less than or equal, greater than, and greater than or equal comparison relations.

x $<<$ **y**
Result of shifting x left by y bits, filling vacated bits with zeros.

```
b ← LENGTH(x)
result ← x
do i = 0 to y-1
   result ← result.bit[1:b-1] || 0b0
```

x $>>$ **y**
Result of shifting x right by y bits, filling vacated bits with copies of bit 0 of x.

```
b ← LENGTH(x)
result ← x
do i = 0 to y-1
   result ← result.bit[0] ||
            result.bit[0:b-2]
```

[]

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 concatented 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.

bcd_CONVERT_FROM_SI128(x,y)

Let x be a signed integer quadword.
Let y indicate the preferred sign code.

Return the signed integer value x in packed decimal format.

```
if x < 0 then do
   x    ← ¬x + 1
   sign ← 0x000D
end
else
        sign ← (y=0) ? 0x000C : 0x000F

result ← 0
shcnt  ← 4

do while (x > 0)
   digit  ← x % 10
   result ← result | (digit<<shcnt)
   x      ← x ÷ 10
   shcnt  ← shcnt + 4
end

return result | sign
```

bcd_INCREMENT(result)

Increments the magnitude of the packed decimal value x by 1.

bcd_SUBTRACT(x,y,z)

Let x and y be 31-digit signed decimal values.

Performs a signed decimal subtract of y from x.

If the unbounded result is equal to zero, eq_flag is set to 1. Otherwise, eq_flag is set to 0.
If the unbounded result is greater than zero, gt_flag is set to 1. Otherwise, gt_flag is set to 0.
If the unbounded result is less than zero, lt_flag is set to 1. Otherwise, lt_flag is set to 0.

If the magnitude of the unbounded result is greater than $10^{31} - 1$, ox_flag is set to 1. Otherwise, ox_flag is set to 0.

If the unbounded result is greater than or equal to zero, the sign code of the result is set to 0b1100 if z=0.
If the unbounded result is greater than or equal to zero, the sign code of the result is set to 0b1111 if z=1.
If the unbounded result is less than zero, the sign code of the result is set to 0b1101.

The low-order 31 digits of the unbounded result magnitude concatented with the sign code are returned.

If either operand is an invalid encoding of a signed decimal value, the result returned is undefined and inv_flag is set to 1 and lt_flag, gt_flag and eq_flag are set to 0. Otherwise, inv_flag is set to 0.

bfp32_ADD(x,y)

x is a binary floating-point value represented in single-precision format.
y is a binary floating-point value represented in single-precision format.
If x is a QNaN, the result is x.
Otherwise, if x is an SNaN, the result is x converted to a QNaN.
Otherwise, if y is a QNaN, the result is y.
Otherwise, if y is an SNaN, the result is y converted to a QNaN.
Otherwise, if x and y are Infinities having opposite signs, the result is the single-precision standard QNaN.
Otherwise, if x is an Infinity, the result is x.
Otherwise, if y is an Infinity, the result is y.
Otherwise, the result is the sum, x added to y, rounded to the nearest single-precision value.
Return the result represented in single-precision format.

bfp32_CONVERT_FROM_SI32(x,y)

Let x be a 32-bit signed integer value.

```
sign           ← X.bit[0]
exp            ← 32 + 127
frac.bit[0]    ← x.bit[0]
frac.bit[1:32] ← x.bit[0:31]

if frac=0 return 0x0000_0000     // Zero operand
if sign=1 then frac = ¬frac + 1

do while (frac.bit[0]=0)
   frac ← frac << 1
   exp  ← exp - 1
end

lsb  ← frac.bit[23]
gbit ← frac.bit[24]
xbit ← frac.bit[25:32]!=0
inc  ← (lsb & gbit) | (gbit & xbit)

frac.bit[0:23] ← frac.bit[0:23] + inc
if carry_out=1 then exp ← exp + 1

result.bit[0]    ← sign
result.bit[1:8]  ← exp - y
result.bit[9:31] ← frac.bit[1:23]

return result
```

bfp32_CONVERT_FROM_UI32(x,y)

x is a 32-bit unsigned integer value.

```
exp  ← 31 + 127
frac ← x.bit[0:31]

if frac=0 return 0x0000_0000 // Zero

do while frac0=0
   frac ← frac << 1
   exp  ← exp - 1
end

lsb  ← frac.bit[23]
gbit ← frac.bit[24]
xbit ← frac.bit[25:31]!=0
inc  ← (lsb & gbit) | (gbit & xbit)
```

```
frac.bit[0:23] ← frac.bit[0:23] + inc
if carry_out=1 then exp ← exp + 1

result.bit[0]    ← 0b0
result.bit[1:8]  ← exp - y
result.bit[9:31] ← frac.bit[1:23]

return result
```

bfp32_LOG_BASE2_ESTIMATE(x)

x is a floating-point value represented in single-precision format.

Returns a floating-point estimate of the base 2 logarithm of x, represented in single-precision format.

bfp32_MAXIMUM(x,y)

x is a floating-point value represented in single-precision format.
y is a floating-point value represented in single-precision format.

Return the largest value of x and y, represented in single-precision format.

The maximum of +0.0 and −0.0 is +0.0.

The maximum of any value and a NaN is a QNaN.

bfp32_MINIMUM(x,y)

x is a floating-point value represented in single-precision format.
y is a floating-point value represented in single-precision format.

Return the smallest value of x and y, represented in single-precision format.

The minimum of +0.0 and −0.0 is −0.0.

The minimum of any value and a NaN is a QNaN.

bfp32_MULTIPLY_ADD(x,z,y)

x is a binary floating-point value represented in single-precision format.
y is a binary floating-point value represented in single-precision format.
z is a binary floating-point value represented in single-precision format.

If x is a QNaN, return x.
Otherwise, if x is an SNaN, the result is x converted to a QNaN.
Otherwise, if y is a QNaN, the result is y.
Otherwise, if y is an SNaN, the result is y converted to a QNaN.
Otherwise, if z is a QNaN, the result is z.
Otherwise, if z is an SNaN, the result is z converted to a QNaN.
Otherwise, if x is an Infinity and z is a Zero, the result is the single-precision standard QNaN.
Otherwise, if x is a Zero and z is an Infinity, the result is the single-precision standard QNaN.
Otherwise, if the product, x multiplied by z, and y are Infinities having opposite signs, the result is the single-precision standard QNaN.
Otherwise, the result is the sum of the product, x multiplied by z, added to y, rounded to the nearest single-precision value.

Return the result represented in single-precision format.

bfp32_NEGATIVE_MULTIPLY_SUBTRACT(x,z,y)

x is a binary floating-point value represented in single-precision format.
y is a binary floating-point value represented in single-precision format.
z is a binary floating-point value represented in single-precision format.

If x is a QNaN, the result is x.
Otherwise, if x is an SNaN, the result is x converted to a QNaN.
Otherwise, if y is a QNaN, the result is y.
Otherwise, if y is an SNaN, the result is y converted to a QNaN.
Otherwise, if z is a QNaN, the result is z.
Otherwise, if z is an SNaN, the result is z converted to a QNaN.
Otherwise, if x is an Infinity and z is a Zero, the result is the single-precision standard QNaN.

Otherwise, if x is a Zero and z is an Infinity, the result is the single-precision standard QNaN.
Otherwise, if the product, x multiplied by z, and y are Infinities having the same signs, the result is the single-precision standard QNaN.
Otherwise, the result is the difference of the product, x multiplied by z, subtracted by y, then rounded to the nearest single-precision value, and then negated.
Return the result represented in single-precision format.

bfp32_POWER2_ESTIMATE(x)
x is a floating-point value represented in single-precision format.
Returns a floating-point estimate of 2 raised to the power of x, represented in single-precision format.

bfp32_RECIPROCAL_ESTIMATE(x)
x is a floating-point value represented in single-precision format.
Returns a floating-point estimate of the reciprocal of x, represented in single-precision format.

bfp32_RECIPROCAL_SQRT_ESTIMATE(x)
x is a floating-point value represented in single-precision format.
Returns a floating-point estimate of the reciprocal of the square root of x, represented in single-precision format.

bfp32_ROUND_TO_INTEGER_CEIL(x)
x is a floating-point value represented in single-precision format.
Returns the smallest floating-point integer that is greater than or equal to x, represented in single-precision format.

bfp32_ROUND_TO_INTEGER_FLOOR(x)
x is a floating-point value represented in single-precision format.
Returns the largest floating-point integer that is less than or equal to x, represented in single-precision format.

bfp32_ROUND_TO_INTEGER_NEAR(x)
x is a floating-point value represented in single-precision format.
Returns the floating-point integer that is nearest to x (in case of a tie, the even single-precision floating-point integer is used), represented in single-precision format.

bfp32_ROUND_TO_INTEGER_TRUNC(x)
x is a floating-point value represented in single-precision format.
Returns the largest floating-point integer that is less than or equal to x if x>0, or the smallest floating-point integer that is greater than or equal to x if x>0, or represented in single-precision format.

bfp32_ROUND_TO_NEAR(x)
x is a floating-point value represented in the working floating-point format.
Returns the single-precision floating-point value that is nearest to x (in case of a tie, the single-precision floating-point value with the least-significant bit equal to 0 is used), represented in single-precision format.

bfp32_SUBTRACT(x,y)
x is a binary floating-point value represented in single-precision format.
y is a binary floating-point value represented in single-precision format.
If x is a QNaN, the result is x.
Otherwise, if x is an SNaN, the result is x converted to a QNaN.
Otherwise, if y is a QNaN, the result is y.
Otherwise, if y is an SNaN, the result is y converted to a QNaN.
Otherwise, if x and y are infinities having the same signs, the result is the single-precision standard QNaN.
Otherwise, if x is an infinity, the result is x.
Otherwise, if y is an infinity, the result is y.
Otherwise, the result is the difference, x subtracted by y, rounded to the nearest single-precision value.
Return the result respresented in single-precision format.

bool_COMPARE_GE_BFP32(x,y)

x is a floating-point value represented in the single-precision format. y is a floating-point value represented in the single-precision format.

Returns the value 1 if x is greater than or equal to y. Otherwise, returns the value 0.

bool_COMPARE_GT_BFP32(x,y)

x is a floating-point value represented in the single-precision format.
y is a floating-point value represented in the single-precision format.

Returns the value 1 if x is greater than y. Otherwise, returns the value 0.

bool_COMPARE_EQ_BFP32(x,y)

x is a floating-point value represented in the single-precision format.
y is a floating-point value represented in the single-precision format.

Returns the value 1 if x is equal to y. Otherwise, returns the value 0.

bool_COMPARE_LE_BFP32(x,y)

x is a floating-point value represented in the single-precision format.
y is a floating-point value represented in the single-precision format.

Returns the value 1 if x is less than or equal to y. Otherwise, returns the value 0.

CHOP8(x)

Returns rightmost 8 bits of x padded on the left with zeros if necessary.

CHOP16(x)

Returns rightmost 16 bits of x padded on the left with zeros if necessary.

CHOP32(x)

Returns rightmost 32 bits of x padded on the left with zeros if necessary.

CHOP64(x)

Returns rightmost 64 bits of x padded on the left with zeros if necessary.

CHOP128(x)

Returns rightmost 128 bits of x padded on the left with zeros if necessary.

Clamp(x,y,z)

x is interpreted as a signed integer. If the value of x is less than y, then the value y is returned, else if the value of x is greater than z, the value z is returned, else the value x is returned.

```
if x < y then
   result ← y
else if x > z then
   result ← z
else
   result ← x
```

EXTS(x)

Result of extending x on the left with copies of bit 0 of x to form a signed integer value having unbounded range.

EXTS8(x)

Result of extending x on the left with copies of bit 0 of x to form an 8-bit signed integer value.

EXTS16(x)

Result of extending x on the left with copies of bit 0 of x to form a 16-bit signed integer value.

EXTS32(x)
Result of extending x on the left with copies of bit 0 of x to form a 32-bit signed integer value.

EXTS64(x)
Result of extending x on the left with copies of bit 0 of x to form a 64-bit signed integer value.

EXTS128(x)
Result of extending x on the left with copies of bit 0 of x to form a 128-bit signed integer value.

EXTZ(x)
Result of extending x on the left with 0s to form a positive signed integer value having unbounded range.

EXTZ8(x)
Result of extending x on the left with 0s to form an 8-bit unsigned integer value.

EXTZ16(x)
Result of extending x on the left with 0s to form a 16-bit unsigned integer value.

EXTZ32(x)
Result of extending x on the left with 0s to form a 32-bit unsigned integer value.

EXTZ64(x)
Result of extending x on the left with 0s to form a 64-bit unsigned integer value.

EXTZ128(x)
Result of extending x on the left with 0s to form a 128-bit unsigned integer value.

InvMixColumns(x)

```
do c = 0 to 3
   result.word[c].byte[0] = 0x0E•x.word[c].byte[0] ⊕ 0x0B•x.word[c].byte[1] ⊕
                            0x0D•x.word[c].byte[2] ⊕ 0x09•x.word[c].byte[3]
   result.word[c].byte[1] = 0x09•x.word[c].byte[0] ⊕ 0x0E•x.word[c].byte[1] ⊕
                            0x0B•x.word[c].byte[2] ⊕ 0x0D•x.word[c].byte[3]
   result.word[c].byte[2] = 0x0D•x.word[c].byte[0] ⊕ 0x09•x.word[c].byte[1] ⊕
                            0x0E•x.word[c].byte[2] ⊕ 0x0B•x.word[c].byte[3]
   result.word[c].byte[3] = 0x0B•x.word[c].byte[0] ⊕ 0x0D•x.word[c].byte[1] ⊕
                            0x09•x.word[c].byte[2] ⊕ 0x0E•x.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.

```
product.bit[0] = x.bit[0] ⊕ x.bit[3]
product.bit[1] = x.bit[1] ⊕ x.bit[4] ⊕ x.bit[0]
product.bit[2] = x.bit[2] ⊕ x.bit[5] ⊕ x.bit[0] ⊕ x.bit[1]
product.bit[3] = x.bit[3] ⊕ x.bit[6] ⊕ x.bit[1] ⊕ x.bit[2]
product.bit[4] = x.bit[4] ⊕ x.bit[7] ⊕ x.bit[0] ⊕ x.bit[2]
product.bit[5] = x.bit[5] ⊕ x.bit[0] ⊕ x.bit[1]
product.bit[6] = x.bit[6] ⊕ x.bit[1] ⊕ x.bit[2]
product.bit[7] = x.bit[7] ⊕ x.bit[2]
```

The $GF(2^8)$ multiply of 0x0B•x can be expressed in minimized terms as the following.

```
product.bit[0] = x.bit[0] ⊕ x.bit[1] ⊕ x.bit[3]
product.bit[1] = x.bit[1] ⊕ x.bit[2] ⊕ x.bit[4] ⊕ x.bit[0]
product.bit[2] = x.bit[2] ⊕ x.bit[3] ⊕ x.bit[5] ⊕ x.bit[0] ⊕ x.bit[1]
product.bit[3] = x.bit[3] ⊕ x.bit[4] ⊕ x.bit[6] ⊕ x.bit[0] ⊕ x.bit[1] ⊕ x.bit[2]
product.bit[4] = x.bit[4] ⊕ x.bit[5] ⊕ x.bit[7] ⊕ x.bit[2]
product.bit[5] = x.bit[5] ⊕ x.bit[6] ⊕ x.bit[0] ⊕ x.bit[1]
product.bit[6] = x.bit[6] ⊕ x.bit[7] ⊕ x.bit[0] ⊕ x.bit[1] ⊕ x.bit[2]
product.bit[7] = x.bit[7] ⊕ x.bit[0] ⊕ x.bit[2]
```

The $GF(2^8)$ multiply of 0x0D•x can be expressed in minimized terms as the following.

```
product.bit[0] = x.bit[0] ⊕ x.bit[2] ⊕ x.bit[3]
product.bit[1] = x.bit[1] ⊕ x.bit[3] ⊕ x.bit[4] ⊕ x.bit[0]
product.bit[2] = x.bit[2] ⊕ x.bit[4] ⊕ x.bit[5] ⊕ x.bit[1]
product.bit[3] = x.bit[3] ⊕ x.bit[5] ⊕ x.bit[6] ⊕ x.bit[0] ⊕ x.bit[2]
product.bit[4] = x.bit[4] ⊕ x.bit[6] ⊕ x.bit[7] ⊕ x.bit[0] ⊕ x.bit[1] ⊕ x.bit[2]
product.bit[5] = x.bit[5] ⊕ x.bit[7] ⊕ x.bit[1]
product.bit[6] = x.bit[6] ⊕ x.bit[0] ⊕ x.bit[2]
product.bit[7] = x.bit[7] ⊕ x.bit[1] ⊕ x.bit[2]
```

The $GF(2^8)$ multiply of 0x0E•x can be expressed in minimized terms as the following.

```
product.bit[0] = x.bit[1] ⊕ x.bit[2] ⊕ x.bit[3]
product.bit[1] = x.bit[2] ⊕ x.bit[3] ⊕ x.bit[4] ⊕ x.bit[0]
product.bit[2] = x.bit[3] ⊕ x.bit[4] ⊕ x.bit[5] ⊕ x.bit[1]
product.bit[3] = x.bit[4] ⊕ x.bit[5] ⊕ x.bit[6] ⊕ x.bit[2]
product.bit[4] = x.bit[5] ⊕ x.bit[6] ⊕ x.bit[7] ⊕ x.bit[1] ⊕ x.bit[2]
product.bit[5] = x.bit[6] ⊕ x.bit[7] ⊕ x.bit[1]
product.bit[6] = x.bit[7] ⊕ x.bit[2]
product.bit[7] = x.bit[0] ⊕ x.bit[1] ⊕ x.bit[2]
```

InvShiftRows(x)

```
result.word[0].byte[0] = x.word[0].byte[0]
result.word[1].byte[0] = x.word[1].byte[0]
result.word[2].byte[0] = x.word[2].byte[0]
result.word[3].byte[0] = x.word[3].byte[0]

result.word[0].byte[1] = x.word[3].byte[1]
result.word[1].byte[1] = x.word[0].byte[1]
result.word[2].byte[1] = x.word[1].byte[1]
result.word[3].byte[1] = x.word[2].byte[1]

result.word[0].byte[2] = x.word[2].byte[2]
result.word[1].byte[2] = x.word[3].byte[2]
result.word[2].byte[2] = x.word[0].byte[2]
result.word[3].byte[2] = x.word[1].byte[2]

result.word[0].byte[3] = x.word[1].byte[3]
result.word[1].byte[3] = x.word[2].byte[3]
result.word[2].byte[3] = x.word[3].byte[3]
result.word[3].byte[3] = x.word[0].byte[3]

return(result)
```

InvSubBytes(x)

```
InvSBOX.byte[256] = {
    0x52,0x09,0x6A,0xD5,0x30,0x36,0xA5,0x38,0xBF,0x40,0xA3,0x9E,0x81,0xF3,0xD7,0xFB,
    0x7C,0xE3,0x39,0x82,0x9B,0x2F,0xFF,0x87,0x34,0x8E,0x43,0x44,0xC4,0xDE,0xE9,0xCB,
    0x54,0x7B,0x94,0x32,0xA6,0xC2,0x23,0x3D,0xEE,0x4C,0x95,0x0B,0x42,0xFA,0xC3,0x4E,
    0x08,0x2E,0xA1,0x66,0x28,0xD9,0x24,0xB2,0x76,0x5B,0xA2,0x49,0x6D,0x8B,0xD1,0x25,
```

```
    0x72,0xF8,0xF6,0x64,0x86,0x68,0x98,0x16,0xD4,0xA4,0x5C,0xCC,0x5D,0x65,0xB6,0x92,
    0x6C,0x70,0x48,0x50,0xFD,0xED,0xB9,0xDA,0x5E,0x15,0x46,0x57,0xA7,0x8D,0x9D,0x84,
    0x90,0xD8,0xAB,0x00,0x8C,0xBC,0xD3,0x0A,0xF7,0xE4,0x58,0x05,0xB8,0xB3,0x45,0x06,
    0xD0,0x2C,0x1E,0x8F,0xCA,0x3F,0x0F,0x02,0xC1,0xAF,0xBD,0x03,0x01,0x13,0x8A,0x6B,
    0x3A,0x91,0x11,0x41,0x4F,0x67,0xDC,0xEA,0x97,0xF2,0xCF,0xCE,0xF0,0xB4,0xE6,0x73,
    0x96,0xAC,0x74,0x22,0xE7,0xAD,0x35,0x85,0xE2,0xF9,0x37,0xE8,0x1C,0x75,0xDF,0x6E,
    0x47,0xF1,0x1A,0x71,0x1D,0x29,0xC5,0x89,0x6F,0xB7,0x62,0x0E,0xAA,0x18,0xBE,0x1B,
    0xFC,0x56,0x3E,0x4B,0xC6,0xD2,0x79,0x20,0x9A,0xDB,0xC0,0xFE,0x78,0xCD,0x5A,0xF4,
    0x1F,0xDD,0xA8,0x33,0x88,0x07,0xC7,0x31,0xB1,0x12,0x10,0x59,0x27,0x80,0xEC,0x5F,
    0x60,0x51,0x7F,0xA9,0x19,0xB5,0x4A,0x0D,0x2D,0xE5,0x7A,0x9F,0x93,0xC9,0x9C,0xEF,
    0xA0,0xE0,0x3B,0x4D,0xAE,0x2A,0xF5,0xB0,0xC8,0xEB,0xBB,0x3C,0x83,0x53,0x99,0x61,
    0x17,0x2B,0x04,0x7E,0xBA,0x77,0xD6,0x26,0xE1,0x69,0x14,0x63,0x55,0x21,0x0C,0x7D }

do i = 0 to 15
   result.byte[i] = InvSBOX.byte[x.byte[i]]
end

return(result)
```

MASK128(x,y)

Let x and y be integer values from 0 to 127.

Generate a 128-bit mask that consists of 1-bits from a start bit, x, through and including a stop bit, y, and 0-bits elsewhere.

```
if x <= y then
   mask = all 0s
   mask.bit[x:y] = all 1s
else
   mask = all 1s
   mask.bit[y+1:x-1] = all 0s
return mask
```

MixColumns(x)

```
do c = 0 to 3
   result.word[c].byte[0] = 0x02•x.word[c].byte[0] ⊕ 0x03•x.word[c].byte[1] ⊕
                                 x.word[c].byte[2] ⊕      x.word[c].byte[3]
   result.word[c].byte[1] =      x.word[c].byte[0] ⊕ 0x02•x.word[c].byte[1] ⊕
                            0x03•x.word[c].byte[2] ⊕      x.word[c].byte[3]
   result.word[c].byte[2] =      x.word[c].byte[0] ⊕      x.word[c].byte[1] ⊕
                            0x02•x.word[c].byte[2] ⊕ 0x03•x.word[c].byte[3]
   result.word[c].byte[3] = 0x03•x.word[c].byte[0] ⊕      x.word[c].byte[1] ⊕
                                 x.word[c].byte[2] ⊕ 0x02•x.word[c].byte[3]
end

return(result)
```

The GF(2^8) multiply of 0x02•x can be expressed in minimized terms as the following.

```
product.bit[0] = x.bit[1]
product.bit[1] = x.bit[2]
product.bit[2] = x.bit[3]
product.bit[3] = x.bit[4] ⊕ x.bit[0]
product.bit[4] = x.bit[5] ⊕ x.bit[0]
product.bit[5] = x.bit[6]
product.bit[6] = x.bit[7] ⊕ x.bit[0]
product.bit[7] = x.bit[0]
```

The GF(2^8) multiply of 0x03•x can be expressed in minimized terms as the following.

```
product.bit[0] = x.bit[0] ⊕ x.bit[1]
product.bit[1] = x.bit[1] ⊕ x.bit[2]
product.bit[2] = x.bit[2] ⊕ x.bit[3]
product.bit[3] = x.bit[3] ⊕ x.bit[4] ⊕ x.bit[0]
product.bit[4] = x.bit[4] ⊕ x.bit[5] ⊕ x.bit[0]
product.bit[5] = x.bit[5] ⊕ x.bit[6]
product.bit[6] = x.bit[6] ⊕ x.bit[7] ⊕ x.bit[0]
product.bit[7] = x.bit[7] ⊕ x.bit[0]
```

qword_bit_splat(x)

x is a 1-bit value.

Return the concatenation of 128 copies of x.

ROTL128(x,y)

Let x be a 128-bit integer value.
Let y be an integer value.

Return x rotated left by y bits.

ShiftRows(x)

```
result.word[0].byte[0] = x.word[0].byte[0]
result.word[1].byte[0] = x.word[1].byte[0]
result.word[2].byte[0] = x.word[2].byte[0]
result.word[3].byte[0] = x.word[3].byte[0]

result.word[0].byte[1] = x.word[1].byte[1]
result.word[1].byte[1] = x.word[2].byte[1]
result.word[2].byte[1] = x.word[3].byte[1]
result.word[3].byte[1] = x.word[0].byte[1]

result.word[0].byte[2] = x.word[2].byte[2]
result.word[1].byte[2] = x.word[3].byte[2]
result.word[2].byte[2] = x.word[0].byte[2]
result.word[3].byte[2] = x.word[1].byte[2]

result.word[0].byte[3] = x.word[3].byte[3]
result.word[1].byte[3] = x.word[0].byte[3]
result.word[2].byte[3] = x.word[1].byte[3]
result.word[3].byte[3] = x.word[2].byte[3]

return(result)
```

si8_CLAMP(x)

Let x be a signed integer value.

Return the value x in 8-bit signed integer format.

- If the value of the element is greater than $2^7 - 1$ the result saturates to $2^7 - 1$ and sat_flag is set to 1.
- If the value of the element is less than -2^7 the result saturates to -2^7 and sat_flag is set to 1.

si16_CLAMP(x)

Let x be a signed integer value.

Return the value x in 16-bit signed integer format.

- If the value of the element is greater than $2^{15} - 1$ the result saturates to $2^{15} - 1$ and SAT is set to 1.
- If the value of the element is less than -2^{15} the result saturates to -2^{15} and SAT is set to 1.

si32_CLAMP(x)

Let x be a signed integer value.

Return the value x in 32-bit signed integer format.

- If the value of the element is greater than $2^{31} - 1$ the result saturates to $2^{31} - 1$ and SAT is set to 1.
- If the value of the element is less than -2^{31} the result saturates to -2^{31} and SAT is set to 1.

si32_CONVERT_FROM_BFP32(x,y)

Let x be a single-precision floating-point value.
Let y be an unsigned integer value.

```
sign            ← x.bit[0]
exp             ← x.bit[1:8]
frac.bit[0:22]  ← x.bit[9:31]
frac.bit[23:30] ← 0b0000_0000
if exp=255 & frac!=0 then return 0x0000_0000      // NaN operand
if exp=255 & frac=0 then do                       // infinity operand
   VSCR.SAT ← 1
   return (sign=1) ? 0x8000_0000 : 0x7FFF_FFFF
end
if (exp+y-127) > 30 then do                             // large operand
   VSCR.SAT ← 1
   return (sign=1) ? 0x8000_0000 : 0x7FFF_FFFF
end
if (exp+y-127) < 0 then return 0x0000_0000              // -1.0 < value < 1.0 (value rounds to 0)
significand.bit[0:31]  ← 0x0000_0000
significand.bit[32]    ← 0x1
significand.bit[33:63] ← frac
do i = 1 to 31-(exp+Y-127)
   significand ← significand >> 1
end
return (sign=0) ? CHOP32(significand) : CHOP32(¬significand + 1)
```

si128_CONVERT_FROM_BCD(x)

Let x be a packed decimal value.

Return the value x in 128-bit signed integer format.

```
result ← 0
scale  ← 1
sign   ← x.bit[124:127]
x      ← 0b0000 || x.nibble[0:30]
do while x > 0
   digit  ← x & 0x000F
   result ← result + (digit × scale)
   x      ← 0b0000 || x.nibble[0:30]
   scale  ← scale × 10
end

if sign=0x000B | sign=0x000D then
   result ← ¬result + 1

return result
```

SubBytes(x)

```
SBOX.byte[0:255] = {
     0x63,0x7C,0x77,0x7B,0xF2,0x6B,0x6F,0xC5,0x30,0x01,0x67,0x2B,0xFE,0xD7,0xAB,0x76,
     0xCA,0x82,0xC9,0x7D,0xFA,0x59,0x47,0xF0,0xAD,0xD4,0xA2,0xAF,0x9C,0xA4,0x72,0xC0,
     0xB7,0xFD,0x93,0x26,0x36,0x3F,0xF7,0xCC,0x34,0xA5,0xE5,0xF1,0x71,0xD8,0x31,0x15,
     0x04,0xC7,0x23,0xC3,0x18,0x96,0x05,0x9A,0x07,0x12,0x80,0xE2,0xEB,0x27,0xB2,0x75,
```

```
        0x09,0x83,0x2C,0x1A,0x1B,0x6E,0x5A,0xA0,0x52,0x3B,0xD6,0xB3,0x29,0xE3,0x2F,0x84,
        0x53,0xD1,0x00,0xED,0x20,0xFC,0xB1,0x5B,0x6A,0xCB,0xBE,0x39,0x4A,0x4C,0x58,0xCF,
        0xD0,0xEF,0xAA,0xFB,0x43,0x4D,0x33,0x85,0x45,0xF9,0x02,0x7F,0x50,0x3C,0x9F,0xA8,
        0x51,0xA3,0x40,0x8F,0x92,0x9D,0x38,0xF5,0xBC,0xB6,0xDA,0x21,0x10,0xFF,0xF3,0xD2,
        0xCD,0x0C,0x13,0xEC,0x5F,0x97,0x44,0x17,0xC4,0xA7,0x7E,0x3D,0x64,0x5D,0x19,0x73,
        0x60,0x81,0x4F,0xDC,0x22,0x2A,0x90,0x88,0x46,0xEE,0xB8,0x14,0xDE,0x5E,0x0B,0xDB,
        0xE0,0x32,0x3A,0x0A,0x49,0x06,0x24,0x5C,0xC2,0xD3,0xAC,0x62,0x91,0x95,0xE4,0x79,
        0xE7,0xC8,0x37,0x6D,0x8D,0xD5,0x4E,0xA9,0x6C,0x56,0xF4,0xEA,0x65,0x7A,0xAE,0x08,
        0xBA,0x78,0x25,0x2E,0x1C,0xA6,0xB4,0xC6,0xE8,0xDD,0x74,0x1F,0x4B,0xBD,0x8B,0x8A,
        0x70,0x3E,0xB5,0x66,0x48,0x03,0xF6,0x0E,0x61,0x35,0x57,0xB9,0x86,0xC1,0x1D,0x9E,
        0xE1,0xF8,0x98,0x11,0x69,0xD9,0x8E,0x94,0x9B,0x1E,0x87,0xE9,0xCE,0x55,0x28,0xDF,
        0x8C,0xA1,0x89,0x0D,0xBF,0xE6,0x42,0x68,0x41,0x99,0x2D,0x0F,0xB0,0x54,0xBB,0x16 }

    do i = 0 to 15
       result.byte[i] = SBOX.byte[x.byte[i]]
    end

    return(result)
```

ui8_CLAMP(x)

Let x be a signed integer value.

Return the value x in 8-bit unsigned integer format.

- If the value of the element is greater than $2^8 - 1$ the result saturates to $2^8 - 1$ and SAT is set to 1.
- If the value of the element is less than 0 the result saturates to 0 and SAT is set to 1.

ui16_CLAMP(x)

Let x be a signed integer value.

Return the value x in 16-bit unsigned integer format.

- If the value of the element is greater than $2^{16} - 1$ the result saturates to $2^{16} - 1$ and SAT is set to 1.
- If the value of the element is less than 0 the result saturates to 0 and SAT is set to 1.

ui32_CLAMP(x)

Let x be a signed integer value.

Return the value x in 32-bit unsigned integer format.

- If the value of the element is greater than $2^{32} - 1$ the result saturates to $2^{32} - 1$ and SAT is set to 1.
- If the value of the element is less than 0 the result saturates to 0 and SAT is set to 1.

ui32_CONVERT_FROM_BFP32(x,y)

Let x be a single-precision floating-point value.

Let y be an unsigned integer value.

```
    sign            ← x.bit[0]
    exp             ← x.bit[1:8]
    frac.bit[0:22]  ← x.bit[9:31]
    frac.bit[23:30] ← 0b0000_0000
    if exp=255 & frac!=0 then return 0x0000_0000     // NaN operand
    if exp=255 & frac=0 then do                      // infinity operand
       VSCR.SAT ← 1
       return (sign=1) ? 0x0000_0000 : 0xFFFF_FFFF
    end
    if (exp+y-127)>31 then do                        // large operand
       VSCR.SAT ← 1
       return (sign=1) ? 0x0000_0000 : 0xFFFF_FFFF
    end
    if (exp+y-127) < 0 then return 0x0000_0000       // -1.0 < value < 1.0
                                                     //   value rounds to 0
    if sign=1 then do                                // negative operand
       VSCR.SAT ← 1
       return 0x0000_0000
```

```
end
significand.bit[0:31]  ← 0x0000_0000
significand.bit[32]    ← 0b1
significand.bit[33:63] ← frac
do i = 1 to 31-(exp+Y-127)
   significand = significand >> 1
end
return CHOP32(significand)
```

6.3 Vector Facility Registers

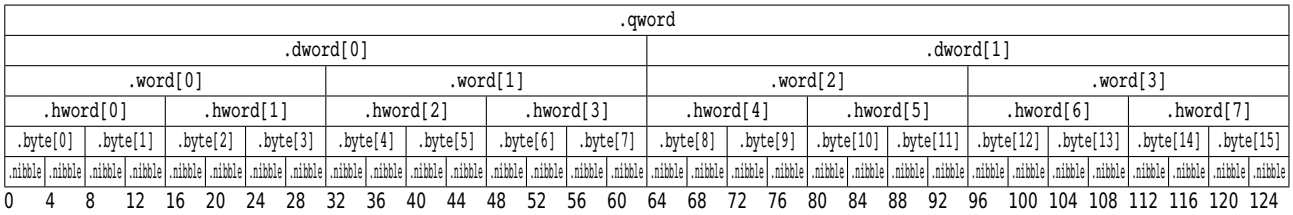


Figure 6.1: Vector-Scalar Register elements

6.3.1 Vector-Scalar Registers

The *Vector* instructions described in Chapter 6 are defined to operate on the higher-numbered 32 Vector-Scalar Registers (VSRs 32-63), formerly known as Vector Registers (VRs 0-31). See Figure 6.2. All computations and other data manipulation are performed on data residing in VSRs 32-63, and results are placed into one of VSRs 32-63.

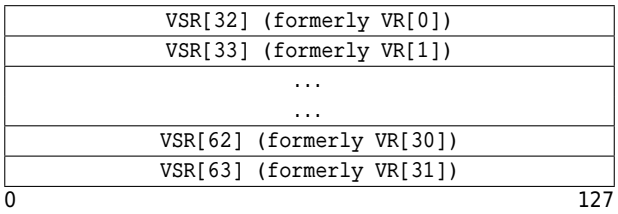


Figure 6.2: Vector-Scalar Registers

Depending on the instruction, the contents of a VSR are interpreted as a sequence of equal-length elements (bytes, halfwords, or words) or as a quadword. Each of the elements is aligned within the VSR, as shown in Figure 6.1. Many instructions perform a given operation in parallel on all elements in a VSR. Depending on the instruction, a byte, halfword, or word element can be interpreted as a signed integer, an unsigned integer, or a logical value; a word element can also be interpreted as a single-precision floating-point value. In the instruction descriptions, phrases like “signed-integer word element” are used as shorthand for “word element, interpreted as a signed integer”.

Load and *Store* instructions are provided that transfer a byte, halfword, word, or quadword between storage and a VSR.

6.3.2 Vector Status and Control Register

The Vector Status and Control Register (VSCR) is a special 32-bit register (not an SPR) that is read and written in a manner similar to the FPSCR in the Power ISA scalar floating-point unit. Special instructions (***mfvscr*** and ***mtvscr***) are provided to move the VSCR from and to a VSR. When moved to or from a VSR, the 32-bit VSCR is right justified in the 128-bit VSR. When moved to a VSR, bits 0:95 of the VSR are cleared (set to 0).

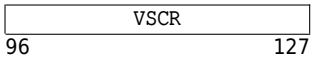


Figure 6.3: Vector Status and Control Register

The bit definitions for the VSCR are as follows.

Bit(s)	Definition
96:110	Reserved
111	**Vector Non-Java Mode** (NJ) This bit controls how denormalized values are handled by *Vector Floating-Point* instructions. 0 Denormalized values are handled as specified by Java and the IEEE standard; see Section 6.6.1. 1 If an element in a source VSR contains a denormalized value, the value 0 is used instead. If an instruction causes an Underflow Exception, the corresponding element in the target VSR is set to 0. In both cases the 0 has the same sign as the denormalized or underflowing value.
112:126	Reserved
127	**Vector Saturation** (SAT) Every vector instruction having “Saturate” in its name implicitly sets this bit to 1 if any result of that instruction “saturates”; see Section 6.8. ***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 ***mtvscr*** instruction.

After the ***mfvscr*** instruction executes, the result in the target VSR 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 ***mfvscr*** 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 ***mtvscr*** is context synchronizing. This implies that all Vector instructions logically preceding an ***mtvscr*** in the program flow will execute in the architectural context (NJ mode) that existed prior to completion of the ***mtvscr***, and that all instructions logically following the ***mtvscr*** will execute in the new context (NJ mode) established by the ***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.

Programming Note

VRSAVE can be used to indicate which VSRs are currently being used by a program. If this is done, the operating system could save only those VSRs when an "interrupt" occurs (see Book III), and could restore only those VSRs when resuming the interrupted program.

If this approach is taken it must be applied rigorously; if a program fails to indicate that a given VSR is in use, software errors may occur that will be difficult to detect and correct because they are timing-dependent.

Some operating systems save and restore VRSAVE only for programs that also use other VSRs.

6.4 Vector Storage Access Operations

The *Vector Storage Access* instructions provide the means by which data can be copied from storage to a VSR or from a VSR to storage. Instructions are provided that access byte, halfword, word, and quadword storage operands. These instructions differ from the fixed-point and floating-point *Storage Access* instructions in that vector storage operands are assumed to be aligned, and vector storage accesses are performed as if the appropriate number of low-order bits of the specified effective address (EA) were zero. For example, the low-order bit of EA is ignored for halfword *Vector Storage Access* instructions, and the low-order four bits of EA are ignored for quadword *Vector Storage Access* instructions. The effect is to load or store the storage operand of the specified length that contains the byte addressed by EA.

If a storage operand is unaligned, additional instructions must be used to ensure that the operand is correctly placed in a VSR or in storage. Instructions are provided that shift and merge the contents of two VSRs, such that an unaligned quadword storage operand can be copied between storage and the VSRs in a relatively efficient manner.

As shown in Figure 6.1, the elements in VSRs are numbered; the high-order (or most significant) byte element is numbered 0 and the low-order (or least significant) byte element is numbered 15. The numbering affects the values that must be placed into the permute control vector for the *Vector Permute* instruction in order for that instruction to achieve the desired effects, as illustrated by the examples in the following subsections.

A vector quadword *Load* instruction for which the effective address (EA) is quadword-aligned places the byte in storage addressed by EA into byte element 0 of the target VSR, the byte in storage addressed by EA+1 into byte element 1 of the target VSR, etc. Similarly, a vector quadword *Store* instruction for which the EA is quadword-aligned places the contents of byte element 0 of the source VSR into the byte in storage addressed by EA, the contents of byte element 1 of the source VSR into the byte in storage addressed by EA+1, etc.

Figure 6.4 shows an aligned quadword in storage. Figure 6.5 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 6.4.

When an aligned byte, halfword, or word storage operand is loaded into a VSR, the element (byte, halfword, or word respectively) that receives the data is the element that would have received the data had the entire aligned quadword containing the storage operand addressed by EA been loaded. Similarly, when a byte, halfword, or word element in a VSR is stored into an aligned storage operand (byte, halfword, or word respectively), the element selected to be stored is the element that would have been stored into the storage operand addressed by EA had the entire VSR been stored to the aligned quadword containing the storage operand addressed by EA. (Byte storage operands are always aligned.)

For aligned byte, halfword, and word storage operands, if the corresponding element number is known when the program is written, the appropriate *Vector Splat* and *Vector Permute* instructions can be used to copy or replicate the data contained in the storage operand after loading the operand into a VSR. An example of this is given in the Programming Note for *Vector Splat*; see page 283. Another example is to replicate the element across an entire VSR before storing it into an arbitrary aligned storage operand of the same length; the replication ensures that the correct data are stored regardless of the offset of the storage operand in its aligned quadword in storage.

6.4.1 Accessing Unaligned Storage Operands

Figure 6.6 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 V*ector Storage Access* instructions ignore the low-order bits of the effective address, the unaligned quadword cannot be transferred between storage and a VSR using a single instruction. The remainder of this section gives examples of accessing unaligned quadword storage operands. Similar sequences can be used to access unaligned halfword and word storage operands.

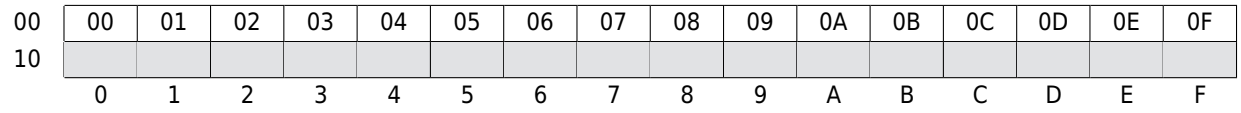


Figure 6.4: Aligned quadword storage operand

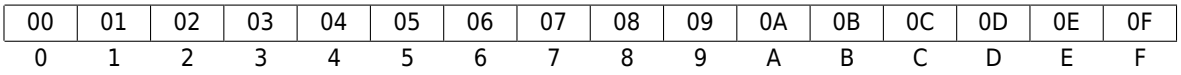


Figure 6.5: VSR contents for aligned quadword Load or Store

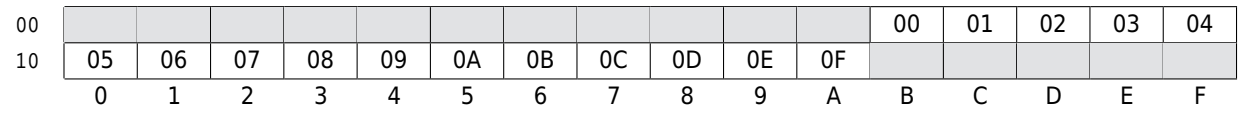


Figure 6.6: Unaligned quadword storage operand

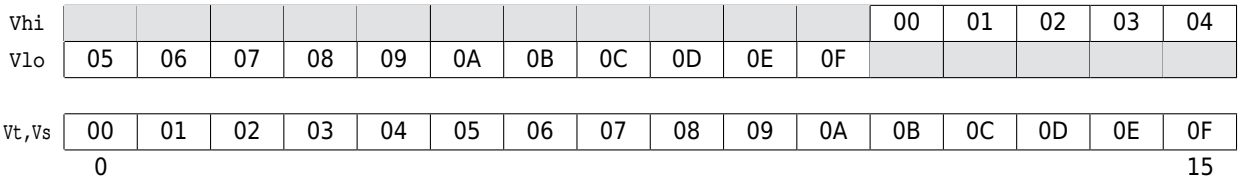


Figure 6.7: VSR contents

Programming Note

The sequence of instructions given below is one approach that can be used to load the unaligned quadword shown in Figure 6.6 into a VSR. In Figure 6.7 Vhi and Vlo are the VSRs that will receive the most significant quadword and least significant quadword respectively. `VSR[VRT+32]` is the target VSR.

After the two quadwords have been loaded into Vhi and Vlo, using *Load Vector Indexed* instructions, the alignment is performed by shifting the 32-byte quantity Vhi || Vlo left by an amount determined by the address of the first byte of the desired data. The shifting is done using a *Vector Permute* instruction for which the permute control vector is generated by a *Load Vector for Shift Left* instruction. The *Load Vector* for *Shift Left* instruction uses the same address specification as the *Load Vector Indexed* instruction that loads the Vhi register; this is the address of the desired unaligned quadword.

The following sequence of instructions copies the unaligned quadword storage operand into register Vt.

```
# Assumptions:
# Rb != 0 and contents of Rb = 0xB
lvx    Vhi,0,Rb       # load MSQ
lvsl   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 6.7 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
lvsr      Vp,0,Rb          # set permute control vector
addi      Rb,Rb,16         # address of LSQ
lvx       Vlo,0,Rb         # load current LSQ
vspltisb  V1s,-1           # generate the select mask bits
vspltisb  V0s,0
vperm     Vmask,V0s,V1s,Vp # generate the select mask
vperm     Vs,Vs,Vs,Vp      # right rotate the data
vsel      Vlo,Vs,Vlo,Vmask # insert LSQ component
vsel      Vhi,Vhi,Vs,Vmask # insert MSQ component
stvx      Vlo,0,Rb         # store LSQ
addi      Rb,Rb,-16        # address of MSQ
stvx      Vhi,0,Rb         # store MSQ
```

6.5 Vector Integer Operations

Many of the instructions that produce fixed-point integer results have the potential to compute a result value that cannot be represented in the target format. When this occurs, this unrepresentable intermediate value is converted to a representable result value using one of the following methods.

1. The high-order bits of the intermediate result that do not fit in the target format are discarded. This method is used by instructions having names that include the word “Modulo”.
2. The intermediate result is converted to the nearest value that is representable in the target format (i.e., to the minimum or maximum representable value, as appropriate). This method is used by instructions having names that include the word “Saturate”. An intermediate result that is forced to the minimum or maximum representable value as just described is said to “saturate”.

   An instruction for which an intermediate result saturates causes 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 SAT is sticky, it can be used to detect whether any instruction in a sequence of “Saturate”-type instructions produced an inexact result due to saturation. For example, the contents of the VSCR can be copied to a VSR (***mfvscr***), bits other than SAT can be cleared in the VSR (***vand*** with a constant), the result can be compared to zero setting CR6 (***vcmpequb***.), and a branch can be taken according to whether SAT was set to 1 (*Branch Conditional* that tests CR field 6).

Testing 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 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 SAT to 1 if any result saturates.)

6.5.1 Integer Saturation

Saturation occurs whenever the result of a saturating instruction does not fit in the result field. Unsigned saturation clamps results to zero (0) on underflow and to the maximum positive integer value ($2^n - 1$, e.g. 255 for byte fields) on overflow. Signed saturation clamps results to the smallest representable negative number (-2^{n-1}, e.g. -128 for byte fields) on underflow, and to the largest representable positive number ($2^{n-1} - 1$, e.g. +127 for byte fields) on overflow.

In most cases, the simple maximum/minimum saturation performed by the vector instructions is adequate. However, sometimes, e.g. in the creation of very high quality images, more complex saturation functions must be applied. To support this, the Vector facility provides a mechanism for detecting that saturation has occurred. The VSCR has a bit, SAT, which is set to a one (1) anytime any field in a saturating instruction saturates. SAT can only be cleared by explicitly writing zero to it. Thus SAT accumulates a summary result of any integer overflow or underflow that occurs on a saturating instruction.

Borderline cases that generate results equal to saturation values, for example unsigned 0+0=0 and unsigned byte 1+254=255, are not considered saturation conditions and do not cause SAT to be set.

SAT can be set by the following types of instructions:

- *Move To VSCR*
- *Vector Add Integer with Saturation*
- *Vector Subtract Integer with Saturation*
- *Vector Multiply-Add Integer with Saturation*
- *Vector Multiply-Sum with Saturation*
- *Vector Sum-Across with Saturation*
- *Vector Pack with Saturation*
- *Vector Convert to Fixed-point with Saturation*

Note that only instructions that explicitly call for “saturation” can set SAT. “Modulo” integer instructions and floating-point arithmetic instructions never set SAT.

Programming Note

The SAT state can be tested and used to alter program flow by moving the VSCR to a VSR (with ***mfvscr***), then masking out bits 0:126 (to clear undefined and reserved bits) and performing a vector compare equal-to unsigned byte w/record (***vcmpequb***.) with zero to get a testable value into the condition register for consumption by a subsequent branch.

Since ***mfvscr*** will be slow compared to other Vector instructions, reading and testing SAT after each instruction would be prohibitively expensive. Therefore, software is advised to employ strategies that minimize checking SAT. For example: checking SAT periodically and backtracking to the last checkpoint to identify exactly which field in which instruction saturated; or, working in an element size sufficient to prevent any overflow or underflow during intermediate calculations, then packing down to the desired element size as the final operation (the vector pack instruction saturates the results and updates SAT when a loss of significance is detected).

6.6 Vector Floating-Point Operations

6.6.1 Floating-Point Overview

Unless NJ=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 4 "Floating-Point Facility" on page 133 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 SAT may be set by the *Vector Convert To Fixed-Point Word* instructions.
- With the exception of the two *Vector Convert To Fixed-Point Word* instructions and three of the four *Vector Round to Floating-Point Integer* instructions, all vector floating-point instructions that round use the rounding mode Round to Nearest.
- Floating-point exceptions (see Section 6.6.2) cannot cause the system error handler to be invoked.

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.

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 NJ=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 NJ=1.
- Exceptions that can be caused by a nonzero source value cannot be caused by a denormalized source value when NJ=1.

6.6.2.1 NaN Operand Exception

A NaN Operand Exception occurs when a source value for any of the following instructions is a NaN.

- A vector instruction that would normally produce floating-point results
- Either of the two *Vector Convert To Fixed-Point Word* instructions
- Any of the four *Vector Floating-Point Compare* instructions

The following actions are taken:

If the vector instruction would normally produce floating-point results, the corresponding result is a source NaN selected as follows. In all cases, if the selected source NaN is a Signaling NaN it is converted to the corresponding Quiet NaN (by setting the high-order bit of the fraction field to 1) before being placed into the target element.

```
if the element in VSR[VRA+32] is a NaN
    then the result is that NaN
    else if the element in VSR[VRB+32] is a NaN
        then the result is that NaN
        else if the element in VSR[VRC+32] is a NaN
            then the result is that NaN
            else if Invalid Operation exception
             (Section 6.6.2.2)
                then the result is the QNaN 0x7FC0_0000
```

If the instruction is either of the two *Vector Convert To Fixed-Point Word* instructions, the corresponding result is 0x0000_0000. SAT is not affected.

If the instruction is *Vector Compare Bounds Floating-Point*, the corresponding result is 0xC000_0000.

If the instruction is one of the other *Vector Floating-Point Compare* instructions, the corresponding result is 0x0000_0000.

6.6.2.2 Invalid Operation Exception

An Invalid Operation Exception occurs when a source value or set of source values is invalid for the specified operation. The invalid operations are:

- Magnitude subtraction of infinities
- Multiplication of infinity by zero

- Reciprocal square root estimate of a negative, nonzero number or −infinity.
- Log base 2 estimate of a negative, nonzero number or −infinity.

The corresponding result is the QNaN 0x7FC0_0000.

6.6.2.3 Zero Divide Exception

A Zero Divide Exception occurs when a *Vector Reciprocal Estimate Floating-Point* or *Vector Reciprocal Square Root Estimate Floating-Point* instruction is executed with a source value of zero.

The corresponding result is an infinity, where the sign is the sign of the source value.

6.6.2.4 Log of Zero Exception

A Log of Zero Exception occurs when a *Vector Log Base 2 Estimate Floating-Point* instruction is executed with a source value of zero.

The corresponding result is −Infinity.

6.6.2.5 Overflow Exception

An Overflow Exception occurs under either of the following conditions.

- For a vector instruction that would normally produce floating-point results, the magnitude of what would have been the result if the exponent range were unbounded exceeds that of the largest finite floating-point number for the target floating-point format.
- For either of the two *Vector Convert To Fixed-Point Word* instructions, either a source value is an infinity or the product of a source value and 2^{UIM} is a number too large in magnitude to be represented in the target fixed-point format.

The following actions are taken:

1. If the vector instruction would normally produce floating-point results, the corresponding result is an infinity, where the sign is the sign of the intermediate result.
2. If the instruction is *Vector Convert To Unsigned Fixed-Point Word Saturate*, the corresponding result is 0xFFFF_FFFF if the source value is a positive number or +infinity, and is 0x0000_0000 if the source value is a negative number or −infinity. SAT is set to 1.
3. If the instruction is *Vector Convert To Signed Fixed-Point Word Saturate*, the corresponding result is 0x7FFF_FFFF if the source value is a positive number or +infinity., and is 0x8000_0000 if the source value is a negative number or −infinity. SAT is set to 1.

6.6.2.6 Underflow Exception

An Underflow Exception can occur only for vector instructions that would normally produce floating-point results. It is detected before rounding. It occurs when a nonzero intermediate result computed as though both the precision and the exponent range were unbounded is less in magnitude than the smallest normalized floating-point number for the target floating-point format.

The following actions are taken:

1. If NJ=0, the corresponding result is the value produced by denormalizing and rounding the intermediate result.
2. If NJ=1, the corresponding result is a zero, where the sign is the sign of the intermediate result.

6.7 Vector Storage Access Instructions

The *Vector Storage Access* instructions compute the effective address (EA) of the storage to be accessed as described in Section 1.10.3, “Effective Address Calculation” on page 31. The low-order bits of the EA that would correspond to an unaligned storage operand are ignored.

The *Load Vector Element Indexed* and *Store Vector Element Indexed* instructions transfer a byte, halfword, or word element between storage and a VSR. The *Load Vector Indexed* and *Store Vector Indexed* instructions transfer an aligned quadword between storage and a VSR.

6.7.1 Storage Access Exceptions

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

6.7.2 Vector Load Instructions

The aligned byte, halfword, word, or quadword in storage addressed by EA is loaded into VSR[VRT+32].

Programming Note

The *Load Vector Element* instructions load the specified element into the same location in the target register as the location into which it would be loaded using the *Load Vector* instruction.

Load Vector Element Byte Indexed X-form

lvebx VRT,RA,RB

31	VRT	RA	RB	7	/
0	6	11	16	21	31

```
if MSR.VEC=0 then Vector_Unavailable()

EA ← ((RA=0) ? 0 : GPR[RA]) + GPR[RB]
eb ← EA.bit[60:63]

VSR[VRT+32] ← undefined
if Big-Endian byte ordering then
   VSR[VRT+32].byte[eb] ← MEM(EA,1)
else
   VSR[VRT+32].byte[15-eb] ← MEM(EA,1)
```

Let 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 the byte in storage at address EA are placed into byte eb of VSR[VRT+32]. The remaining bytes of VSR[VRT+32] are set to undefined values.

If Little-Endian byte ordering is used for the storage access, the contents of the byte in storage at address EA are placed into byte 15−eb of VSR[VRT+32]. The remaining bytes of VSR[VRT+32] are set to undefined values.

Special Registers Altered:

None

Load Vector Element Halfword Indexed X-form

lvehx VRT,RA,RB

31	VRT	RA	RB	39	/
0	6	11	16	21	31

```
if MSR.VEC=0 then Vector_Unavailable()

EA ← ((RA=0) ? 0 : GPR[RA]) + GPR[RB]
EA ← EA & 0xFFFF_FFFF_FFFF_FFFE
eb ← EA.bit[60:63]

VSR[VRT+32] ← undefined
if Big-Endian byte ordering then
   VSR[VRT+32].byte[eb:eb+1] ← MEM(EA,2)
else
   VSR[VRT+32].byte[14-eb:15-eb] ← MEM(EA,2)
```

Let EA be the result of ANDing 0xFFFF_FFFF_FFFF_FFFE with the sum (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 VSR[VRT+32],
- the contents of the byte in storage at address EA+1 are placed into byte eb+1 of VSR[VRT+32], and
- the remaining bytes of VSR[VRT+32] are set to undefined values.

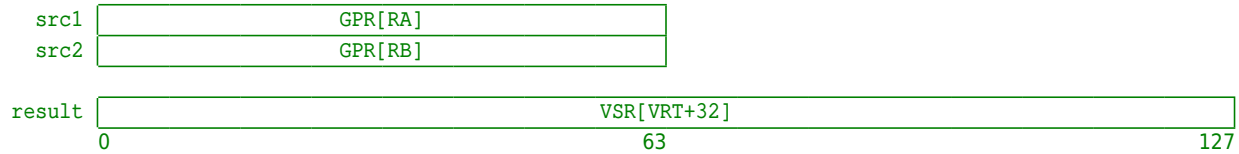
If Little-Endian byte ordering is used for the storage access,

- the contents of the byte in storage at address EA are placed into byte 15−eb of VSR[VRT+32],
- the contents of the byte in storage at address EA+1 are placed into byte 14−eb of VSR[VRT+32], and
- the remaining bytes of VSR[VRT+32] are set to undefined values.

Special Registers Altered:

None

Register Data Layout for lvebx & lvehx



Load Vector Element Word Indexed X-form

lvewx VRT,RA,RB

31	VRT	RA	RB	71	/
0	6	11	16	21	31

```
if MSR.VEC=0 then Vector_Unavailable()

EA ← ((RA=0) ? 0 : GPR[RA]) + GPR[RB]
EA ← EA & 0xFFFF_FFFF_FFFF_FFFC
eb ← EA.bit[60:63]

VSR[VRT+32] ← undefined
if Big-Endian byte ordering then
   VSR[VRT+32].byte[eb:eb+3] ← MEM(EA,4)
else
   VSR[VRT+32].byte[12-eb:15-eb] ← MEM(EA,4)
```

Let 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 VSR[VRT+32],
- the contents of the byte in storage at address EA+1 are placed into byte eb+1 of VSR[VRT+32],
- the contents of the byte in storage at address EA+2 are placed into byte eb+2 of VSR[VRT+32],
- the contents of the byte in storage at address EA+3 are placed into byte eb+3 of VSR[VRT+32], and
- the remaining bytes of VSR[VRT+32] are set to undefined values.

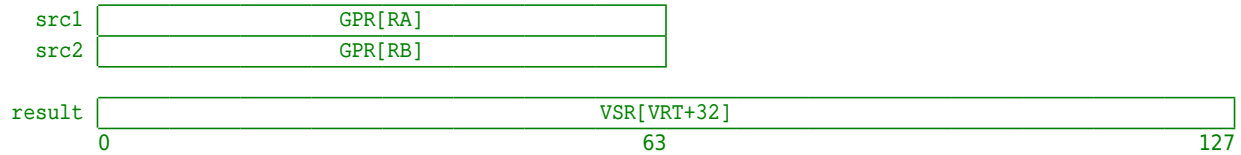
If if Little-Endian byte ordering is used for the storage access,

- the contents of the byte in storage at address EA are placed into byte 15-eb of VSR[VRT+32],
- the contents of the byte in storage at address EA+1 are placed into byte 14-eb of VSR[VRT+32],
- the contents of the byte in storage at address EA+2 are placed into byte 13-eb of VSR[VRT+32],
- the contents of the byte in storage at address EA+3 are placed into byte 12-eb of VSR[VRT+32], and
- the remaining bytes of VSR[VRT+32] are set to undefined values.

Special Registers Altered:

None

Register Data Layout for lvewx



Load Vector Indexed X-form

lvx VRT,RA,RB

31	VRT	RA	RB	103	/
0	6	11	16	21	31

```
if MSR.VEC=0 then Vector_Unavailable()

EA ← ((RA=0) ? 0 : GPR[RA]) + GPR[RB]
EA ← EA & 0xFFFF_FFFF_FFFF_FFF0

VSR[VRT+32] ← MEM(EA, 16)
```

Let EA be the result of ANDing 0xFFFF_FFFF_FFFF_FFF0 with the sum (RA|0) + (RB).

The contents of the quadword in storage at address EA are placed into VSR[VRT+32].

Special Registers Altered:

None

Load Vector Indexed Last X-form

lvxl VRT,RA,RB

31	VRT	RA	RB	359	/
0	6	11	16	21	31

```
if MSR.VEC=0 then Vector_Unavailable()

EA ← ((RA=0) ? 0 : GPR[RA]) + GPR[RB]
EA ← EA & 0xFFFF_FFFF_FFFF_FFF0

VSR[VRT+32] ← MEM(EA, 16)

mark_as_not_likely_to_be_needed_again_anytime_soon(EA)
```

Let EA be the result of ANDing 0xFFFF_FFFF_FFFF_FFF0 with the sum (RA|0) + (RB).

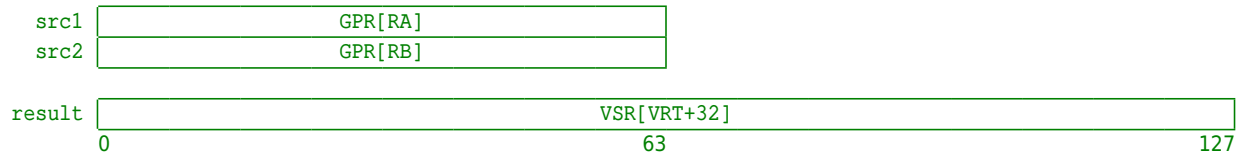
The contents of the quadword in storage at address EA are placed into VSR[VRT+32].

lvxl provides a hint that the quadword in storage addressed by EA will probably not be needed again by the program in the near future.

Special Registers Altered:

None

Register Data Layout for lvx & lvxl



Programming Note

On some implementations, the hint provided by the ***lvxl*** instruction and the corresponding hint provided by the *stvxl* instruction are applied to the entire cache block containing the specified quadword. On such implementations, the effect of the hint may be to cause that cache block to be considered a likely candidate for replacement when space is needed in the cache for a new block. Thus, on such implementations, the hint should be used with caution if the cache block containing the quadword also contains data that may be needed by the program in the near future. Also, the hint may be used before the last reference in a sequence of references to the quadword if the subsequent references are likely to occur sufficiently soon that the cache block containing the quadword is not likely to be displaced from the cache before the last reference.

6.7.3 Vector Store Instructions

Some portion or all of the contents of VSR[VRS+32] are stored into the aligned byte, halfword, word, or quadword in storage addressed by EA.

Programming Note

The *Store Vector Element* instructions store the specified element into the same storage location as the location into which it would be stored using the *Store Vector* instruction.

Store Vector Element Byte Indexed X-form

stvebx VRS,RA,RB

31	VRS	RA	RB	135	/
0	6	11	16	21	31

```
if MSR.VEC=0 then Vector_Unavailable()

EA ← ((RA=0) ? 0 : GPR[RA]) + GPR[RB]
eb ← EA.bit[60:63]

if Big-Endian byte ordering then
   MEM(EA,1) ← VSR[VRS+32].byte[eb]
else
   MEM(EA,1) ← VSR[VRS+32].byte[15-eb]
```

Let 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 VSR[VRS+32] are placed in the byte in storage at address EA.

If Little-Endian byte ordering is used for the storage access, the contents of byte 15-eb of VSR[VRS+32] are placed in the byte in storage at address EA.

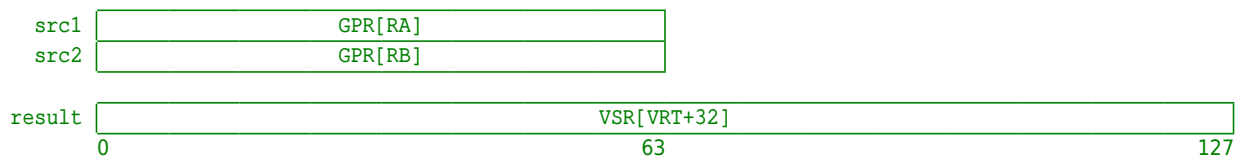
Special Registers Altered:

None

Programming Note

Unless bits 60:63 of the address are known to match the byte offset of the subject byte element in VSR[VRS+32], software should use *Vector Splat* to splat the subject byte element before performing the store.

Register Data Layout for stvebx



Store Vector Element Halfword Indexed X-form

stvehx VRS,RA,RB

31	VRS	RA	RB	167	/
0	6	11	16	21	31

```
if MSR.VEC=0 then Vector_Unavailable()

EA ← ((RA=0) ? 0 : GPR[RA]) + GPR[RB]
EA ← EA & 0xFFFF_FFFF_FFFF_FFFE
eb ← EA.bit[60:63]

if Big-Endian byte ordering then
   MEM(EA,2) ← VSR[VRS+32].byte[eb:eb+1]
else
   MEM(EA,2) ← VSR[VRS+32][14-eb:15-eb]
```

Let EA be the result of ANDing 0xFFFF_FFFF_FFFF_FFFE with the sum (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 VSR[VRS+32] are placed in the byte in storage at address EA, and
- the contents of byte eb+1 of VSR[VRS+32] are placed in the byte in storage at address EA+1.

If Little-Endian byte ordering is used for the storage access,

- the contents of byte 15-eb of VSR[VRS+32] are placed in the byte in storage at address EA, and
- the contents of byte 14-eb of VSR[VRS+32] are placed in the byte in storage at address EA+1.

Special Registers Altered:

None

Programming Note

Unless bits 60:62 of the address are known to match the halfword offset of the subject halfword element in VSR[VRS+32], software should use *Vector Splat* to splat the subject halfword element before performing the store.

Store Vector Element Word Indexed X-form

stvewx VRS,RA,RB

31	VRS	RA	RB	199	/
0	6	11	16	21	31

```
if MSR.VEC=0 then Vector_Unavailable()

EA ← ((RA=0) ? 0 : GPR[RA]) + GPR[RB]
EA ← EA & 0xFFFF_FFFF_FFFF_FFFC
eb ← EA.bit[60:63]

if Big-Endian byte ordering then
   MEM(EA,4) ← VSR[VRS+32].byte[eb:eb+3]
else
   MEM(EA,4) ← VSR[VRS+32].byte[12-eb:15-eb]
```

Let EA be the result of ANDing 0xFFFF_FFFF_FFFF_FFFC with the sum (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 VSR[VRS+32] are placed in the byte in storage at address EA,
- the contents of byte eb+1 of VSR[VRS+32] are placed in the byte in storage at address EA+1,
- the contents of byte eb+2 of VSR[VRS+32] are placed in the byte in storage at address EA+2, and
- the contents of byte eb+3 of VSR[VRS+32] are placed in the byte in storage at address EA+3.

If Little-Endian byte ordering is used for the storage access,

- the contents of byte 15-eb of VSR[VRS+32] are placed in the byte in storage at address EA,
- the contents of byte 14-eb of VSR[VRS+32] are placed in the byte in storage at address EA+1,
- the contents of byte 13-eb of VSR[VRS+32] are placed in the byte in storage at address EA+2, and
- the contents of byte 12-eb of VSR[VRS+32] are placed in the byte in storage at address EA+3.

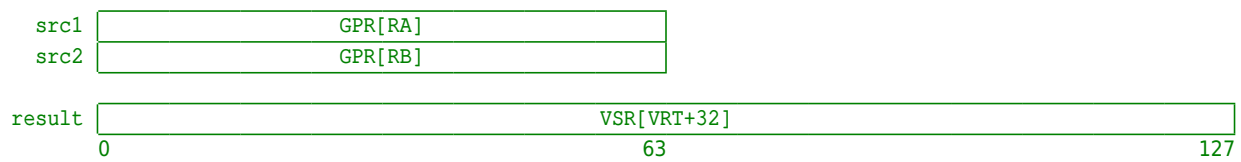
Special Registers Altered:

None

Programming Note

Unless bits 60:61 of the address are known to match the word offset of the subject word element in VSR[VRS+32], software should use *Vector Splat* to splat the subject word element before performing the store.

Register Data Layout for stvehx & stvewx



Store Vector Indexed X-form

stvx VRS,RA,RB

31	VRS	RA	RB	231	/
0	6	11	16	21	31

```
if MSR.VEC=0 then Vector_Unavailable()

EA ← ((RA=0) ? 0 : GPR[RA]) + GPR[RB]
EA ← EA & 0xFFFF_FFFF_FFFF_FFF0

MEM(EA, 16) ← VSR[VRS+32]
```

Let EA be the result of ANDing 0xFFFF_FFFF_FFFF_FFF0 with the sum (RA|0) + (RB).

The contents of VSR[VRS+32] are placed into the quadword in storage at address EA.

Special Registers Altered:

None

Store Vector Indexed Last X-form

stvxl VRS,RA,RB

31	VRS	RA	RB	487	/
0	6	11	16	21	31

```
if MSR.VEC=0 then Vector_Unavailable()

EA ← ((RA=0) ? 0 : GPR[RA]) + GPR[RB]
EA ← EA & 0xFFFF_FFFF_FFFF_FFF0

MEM(EA, 16) ← VSR[VRS+32]

mark_as_not_likely_to_be_needed_again_anytime_soon(EA)
```

Let EA be the result of ANDing 0xFFFF_FFFF_FFFF_FFF0 with the sum (RA|0) + (RB).

The contents of VSR[VRS+32] are placed into the quadword in storage at address EA.

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.

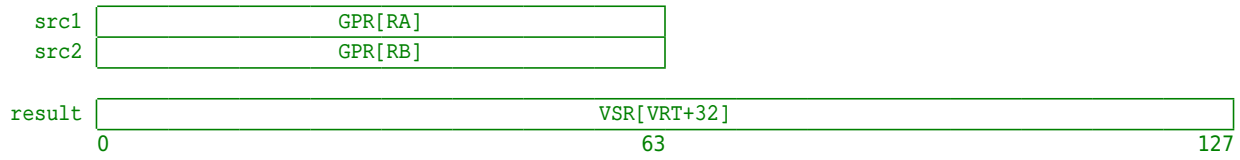
Special Registers Altered:

None

Programming Note

See the Programming Note for the ***lvxl*** instruction on page 262.

Register Data Layout for stvx & stvxl



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 286). Let X and Y be the contents of VSR[VRA+32] and VSR[VRB+32] specified by the ***vperm***. The control vector created by ***lvsl*** causes the ***vperm*** to select the high-order 16 bytes of the result of shifting the 32-byte value X || Y left by 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.

Programming Note

Examples of uses of ***lvsl***, ***lvsr***, and ***vperm*** to load and store unaligned data are given in Section 6.4.1.

These instructions can also be used to rotate or shift the contents of a VSR left (***lvsl***) or right (***lvsr***) by sh bytes. For rotating, the VSR to be rotated should be specified as both VSR[VRA+32] and VSR[VRB+32] for ***vperm***. For shifting left, VSR[VRB+32] for ***vperm*** should be a register containing all zeros and VSR[VRA+32] should contain the value to be shifted, and vice versa for shifting right.

Load Vector for Shift Left Indexed X-form

lvsl VRT,RA,RB

31	VRT	RA	RB	6	/
0	6	11	16	21	31

```
if MSR.VEC=0 then Vector_Unavailable()

sh ← (((RA=0) ? 0 : GPR[RA]) + GPR[RB]).bit[60:63]

switch(sh)
   case(0x0): v←0x000102030405060708090A0B0C0D0E0F
   case(0x1): v←0x0102030405060708090A0B0C0D0E0F10
   case(0x2): v←0x02030405060708090A0B0C0D0E0F1011
   case(0x3): v←0x030405060708090A0B0C0D0E0F101112
   case(0x4): v←0x0405060708090A0B0C0D0E0F10111213
   case(0x5): v←0x05060708090A0B0C0D0E0F1011121314
   case(0x6): v←0x060708090A0B0C0D0E0F101112131415
   case(0x7): v←0x0708090A0B0C0D0E0F10111213141516
   case(0x8): v←0x08090A0B0C0D0E0F1011121314151617
   case(0x9): v←0x090A0B0C0D0E0F101112131415161718
   case(0xA): v←0x0A0B0C0D0E0F10111213141516171819
   case(0xB): v←0x0B0C0D0E0F101112131415161718191A
   case(0xC): v←0x0C0D0E0F10111213141516171819A1B
   case(0xD): v←0x0D0E0F101112131415161718191A1B1C
   case(0xE): v←0x0E0F101112131415161718191A1B1C1D
   case(0xF): v←0x0F101112131415161718191A1B1C1D1E
VSR[VRT+32] ← v
```

Let sh be bits 60:63 of the sum of the contents of GPR[RA], or 0 if RA=0, and the contents of GPR[RB].

Let X be the 32-byte value 0x00 || 0x01 || 0x02 || ... || 0x1D || 0x1E || 0x1F.

Bytes sh to sh+15 of X are placed into VSR[VRT+32].

Special Registers Altered:

None

Load Vector for Shift Right Indexed X-form

lvsr VRT,RA,RB

31	VRT	RA	RB	38	/
0	6	11	16	21	31

```
if MSR.VEC=0 then Vector_Unavailable()

sh ← (((RA=0) ? 0 : GPR[RA]) + GPR[RB]).bit[60:63]

switch(sh)
   case(0x0): v←0x101112131415161718191A1B1C1D1E1F
   case(0x1): v←0x0F101112131415161718191A1B1C1D1E
   case(0x2): v←0x0E0F101112131415161718191A1B1C1D
   case(0x3): v←0x0D0E0F101112131415161718191A1B1C
   case(0x4): v←0x0C0D0E0F101112131415161718191A1B
   case(0x5): v←0x0B0C0D0E0F101112131415161718191A
   case(0x6): v←0x0A0B0C0D0E0F10111213141516171819
   case(0x7): v←0x090A0B0C0D0E0F101112131415161718
   case(0x8): v←0x08090A0B0C0D0E0F1011121314151617
   case(0x9): v←0x0708090A0B0C0D0E0F10111213141516
   case(0xA): v←0x060708090A0B0C0D0E0F101112131415
   case(0xB): v←0x05060708090A0B0C0D0E0F1011121314
   case(0xC): v←0x0405060708090A0B0C0D0E0F10111213
   case(0xD): v←0x030405060708090A0B0C0D0E0F101112
   case(0xE): v←0x02030405060708090A0B0C0D0E0F1011
   case(0xF): v←0x0102030405060708090A0B0C0D0E0F10
VSR[VRT+32] ← v
```

Let sh be bits 60:63 of the sum of the contents of GPR[RA], or 0 if RA=0, and the contents of GPR[RB].

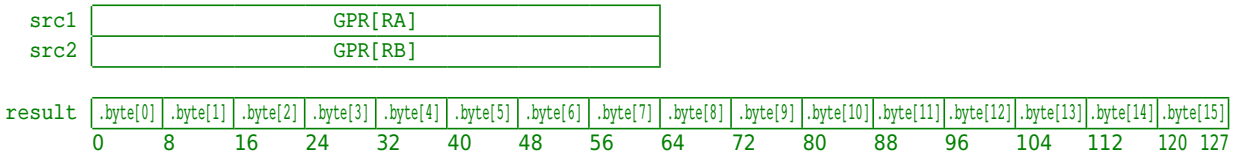
Let X be the 32-byte value 0x00 || 0x01 || 0x02 || ... || 0x1D || 0x1E || 0x1F.

Bytes 16-sh to 31-sh of X are placed into VSR[VRT+32].

Special Registers Altered:

None

Register Data Layout for lvsl & lvsr



6.8 Vector Permute and Formatting Instructions

6.8.1 Vector Pack Instructions

Vector Pack Pixel VX-form

vpkpx VRT,VRA,VRB

4	VRT	VRA	VRB	782
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

vsrc.qword[0] ← VSR[VRA+32]
vsrc.qword[1] ← VSR[VRB+32]

do i = 0 to 7
   VSR[VRT+32].hword[i].bit[0]     ←
               vsrc.word[i].bit[7]
   VSR[VRT+32].hword[i].bit[1:5]   ←
               vsrc.word[i].bit[8:12]
   VSR[VRT+32].hword[i].bit[6:10]  ←
               vsrc.word[i].bit[16:20]
   VSR[VRT+32].hword[i].bit[11:15] ←
               vsrc.word[i].bit[24:28]
end
```

Let vsrc be the concatenation of the contents of VSR[VRA+32] followed by the contents of VSR[VRB+32].

For each integer value i from 0 to 7, do the following.

The contents of word element i of vsrc are 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 VSR[VRT+32].

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.

Register Data Layout for vpkpx

src1	VSR[VRA+32].word[0]	VSR[VRA+32].word[1]	VSR[VRA+32].word[2]	VSR[VRA+32].word[3]
src2	VSR[VRB+32].word[0]	VSR[VRB+32].word[1]	VSR[VRB+32].word[2]	VSR[VRB+32].word[3]

result	VSR[VRT+32].hword[0]	VSR[VRT+32].hword[1]	VSR[VRT+32].hword[2]	VSR[VRT+32].hword[3]	VSR[VRT+32].hword[4]	VSR[VRT+32].hword[5]	VSR[VRT+32].hword[6]	VSR[VRT+32].hword[7]
	0	16	32	48	64	80	96	112 127

Vector Pack Signed Halfword Signed Saturate VX-form

vpkshss VRT,VRA,VRB

4	VRT	VRA	VRB	398
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

vsrc.qword[0] ← VSR[VRA+32]
vsrc.qword[1] ← VSR[VRB+32]

do i = 0 to 15
   VSR[VRT+32].byte[i] ← si8_CLAMP(EXTS(vsrc.hword[i]))
end
```

Let vsrc be the concatenation of the contents of VSR[VRA+32] followed by the contents of VSR[VRB+32].

For each integer value i from 0 to 15, do the following.

The signed integer value in halfword element i of vsrc is placed into byte element i of VSR[VRT+32] in signed integer format.

- If the value of the element is greater than 2^7-1 the result saturates to 2^7-1 and SAT is set to 1.
- If the value of the element is less than -2^7 the result saturates to -2^7 and SAT is set to 1.

Special Registers Altered:

Register	*Field(s)*
VSCR	SAT

Vector Pack Signed Halfword Unsigned Saturate VX-form

vpkshus VRT,VRA,VRB

4	VRT	VRA	VRB	270
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

vsrc.qword[0] ← VSR[VRA+32]
vsrc.qword[1] ← VSR[VRB+32]

do i = 0 to 15
   VSR[VRT+32].byte[i] ← ui8_CLAMP(EXTS(vsrc.hword[i]))
end
```

Let vsrc be the concatenation of the contents of VSR[VRA+32] followed by the contents of VSR[VRB+32].

For each integer value i from 0 to 15, do the following.

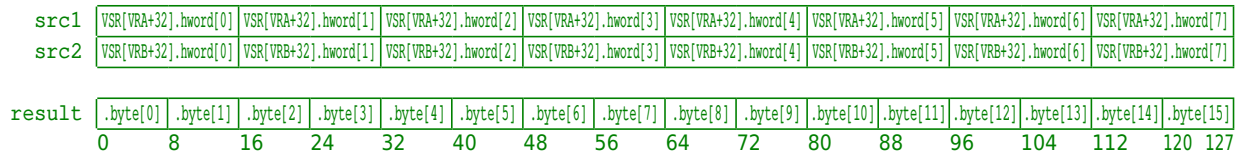
The signed integer value in halfword element i of vsrc is placed into byte element i of VSR[VRT+32] in unsigned integer format.

- If the value of the element is greater than 2^8-1 the result saturates to 2^8-1 and SAT is set to 1.
- If the value of the element is less than 0 the result saturates to 0 and SAT is set to 1.

Special Registers Altered:

Register	*Field(s)*
VSCR	SAT

Register Data Layout for vpkshss & vpkshus



Vector Pack Signed Word Signed Saturate VX-form

vpkswss VRT,VRA,VRB

4	VRT	VRA	VRB	462
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

vsrc.qword[0] ← VSR[VRA+32]
vsrc.qword[1] ← VSR[VRB+32]

do i = 0 to 7
   VSR[VRT+32].hword[i] ←
                si16_CLAMP(EXTS(vsrc.word[i]))
end
```

Let vsrc be the concatenation of the contents of VSR[VRA+32] followed by the contents of VSR[VRB+32].

For each integer value i from 0 to 7, do the following.

The signed integer value in word element i of vsrc is placed into halfword element i of VSR[VRT+32] in signed integer format.

- If the value of the element is greater than $2^{15}-1$ the result saturates to $2^{15}-1$ and SAT is set to 1.
- If the value of the element is less than -2^{15} the result saturates to -2^{15} and SAT is set to 1.

Special Registers Altered:

Register	*Field(s)*
VSCR	SAT

Vector Pack Signed Word Unsigned Saturate VX-form

vpkswus VRT,VRA,VRB

4	VRT	VRA	VRB	334
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

vsrc.qword[0] ← VSR[VRA+32]
vsrc.qword[1] ← VSR[VRB+32]

do i = 0 to 7
   VSR[VRT+32].hword[i] ←
                ui16_CLAMP(EXTS(vsrc.word[i]))
end
```

Let vsrc be the concatenation of the contents of VSR[VRA+32] followed by the contents of VSR[VRB+32].

For each integer value i from 0 to 7, do the following.

The signed integer value in word element i of vsrc is placed into halfword element i of VSR[VRT+32] in unsigned integer format.

- If the value of the element is greater than $2^{16}-1$ the result saturates to $2^{16}-1$ and SAT is set to 1.
- If the value of the element is less than 0 the result saturates to 0 and SAT is set to 1.

Special Registers Altered:

Register	*Field(s)*
VSCR	SAT

Register Data Layout for vpkswss & vpkswus

src1	VSR[VRA+32].word[0]	VSR[VRA+32].word[1]	VSR[VRA+32].word[2]	VSR[VRA+32].word[3]
src2	VSR[VRB+32].word[0]	VSR[VRB+32].word[1]	VSR[VRB+32].word[2]	VSR[VRB+32].word[3]

result	VSR[VRT+32].hword[0]	VSR[VRT+32].hword[1]	VSR[VRT+32].hword[2]	VSR[VRT+32].hword[3]	VSR[VRT+32].hword[4]	VSR[VRT+32].hword[5]	VSR[VRT+32].hword[6]	VSR[VRT+32].hword[7]
	0	16	32	48	64	80	96	112 127

Vector Pack Signed Doubleword Signed Saturate VX-form

vpksdss VRT,VRA,VRB

4	VRT	VRA	VRB	1486
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

VSR[VRT+32].word[0] ←
        si32_CLAMP(EXTS(VSR[VRA+32].dword[0]))
VSR[VRT+32].word[1] ←
        si32_CLAMP(EXTS(VSR[VRA+32].dword[1]))
VSR[VRT+32].word[2] ←
        si32_CLAMP(EXTS(VSR[VRB+32].dword[0]))
VSR[VRT+32].word[3] ←
        si32_CLAMP(EXTS(VSR[VRB+32].dword[1]))
```

Let vsrc be the concatenation of the contents of VSR[VRA+32] followed by the contents of VSR[VRB+32].

For each integer value i from 0 to 3, do the following.

The signed integer value in doubleword element i of vsrc is placed into word element i of VSR[VRT+32] in signed integer format.

- If the value is greater than $2^{31} - 1$ the result saturates to $2^{31} - 1$ and SAT is set to 1.
- If the value is less than -2^{31} the result saturates to -2^{31} and SAT is set to 1.

Special Registers Altered:

Register	*Field(s)*
VSCR	SAT

Vector Pack Signed Doubleword Unsigned Saturate VX-form

vpksdus VRT,VRA,VRB

4	VRT	VRA	VRB	1358
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

VSR[VRT+32].word[0] ←
        ui32_CLAMP(EXTS(VSR[VRA+32].dword[0]))
VSR[VRT+32].word[1] ←
        ui32_CLAMP(EXTS(VSR[VRA+32].dword[1]))
VSR[VRT+32].word[2] ←
        ui32_CLAMP(EXTS(VSR[VRB+32].dword[0]))
VSR[VRT+32].word[3] ←
        ui32_CLAMP(EXTS(VSR[VRB+32].dword[1]))
```

Let vsrc be the concatenation of the contents of VSR[VRA+32] followed by the contents of VSR[VRB+32].

For each integer value i from 0 to 3, do the following.

The signed integer value in doubleword element i of vsrc is placed into word element i of VSR[VRT+32] in unsigned integer format.

- If the value is greater than $2^{32} - 1$ the result saturates to $2^{32} - 1$ and SAT is set to 1.
- If the value is less than 0 the result saturates to 0 and SAT is set to 1.

Special Registers Altered:

Register	*Field(s)*
VSCR	SAT

Register Data Layout for vpksdss & vpksdus

src1	VSR[VRA+32].dword[0]	VSR[VRA+32].dword[1]
src2	VSR[VRB+32].dword[0]	VSR[VRB+32].dword[1]

result	VSR[VRT+32].word[0]	VSR[VRT+32].word[1]	VSR[VRT+32].word[2]	VSR[VRT+32].word[3]
	0	32	64	96 127

Vector Pack Unsigned Halfword Unsigned Modulo VX-form

vpkuhum VRT,VRA,VRB

4	VRT	VRA	VRB	14
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

vsrc.qword[0] ← VSR[VRA+32]
vsrc.qword[1] ← VSR[VRB+32]

do i = 0 to 15
   VSR[VRT+32].byte[i] ←
        vsrc.hword[i].bit[8:15]
end
```

Let vsrc be the concatenation of the contents of VSR[VRA+32] followed by the contents of VSR[VRB+32].

For each integer value i from 0 to 15, do the following.

The contents of bits 8:15 of halfword element i of vsrc are placed into byte element i of VSR[VRT+32].

Special Registers Altered:

None

Vector Pack Unsigned Halfword Unsigned Saturate VX-form

vpkuhus VRT,VRA,VRB

4	VRT	VRA	VRB	142
0	6	11	16	21 31

```
if MSR.VEC then Vector_Unavailable()

vsrc.qword[0] ← VSR[VRA+32]
vsrc.qword[1] ← VSR[VRB+32]

do i = 0 to 15
   VSR[VRT+32].byte[i] ←
        ui8_CLAMP(EXTZ(vsrc.hword[i]))
end
```

Let vsrc be the concatenation of the contents of VSR[VRA+32] followed by the contents of VSR[VRB+32].

For each integer value i from 0 to 15, do the following.

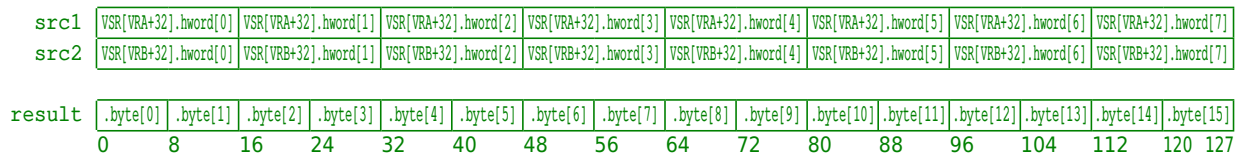
The unsigned integer value in halfword element i of vsrc are placed into byte element i of VSR[VRT+32] in unsigned integer format.

- If the value of the element is greater than $2^8 - 1$ the result saturates to $2^8 - 1$ and SAT is set to 1.

Special Registers Altered:

Register	*Field(s)*
VSCR	SAT

Register Data Layout for vpkuhum & vpkuhus



Vector Pack Unsigned Word Unsigned Modulo VX-form

vpkuwum VRT,VRA,VRB

4	VRT	VRA	VRB	78
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

vsrc.qword[0] ← VSR[VRA+32]
vsrc.qword[1] ← VSR[VRB+32]

do i = 0 to 7
   VSR[VRT+32].hword[i] ← vsrc.word[i].bit[16:31]
end
```

Let vsrc be the concatenation of the contents of VSR[VRA+32] followed by the contents of VSR[VRB+32].

For each integer value i from 0 to 7, do the following.

The contents of bits 16:31 of word element i of vsrc are placed into halfword element i of VSR[VRT+32].

Special Registers Altered:

None

Vector Pack Unsigned Word Unsigned Saturate VX-form

vpkuwus VRT,VRA,VRB

4	VRT	VRA	VRB	206
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

vsrc.qword[0] ← VSR[VRA+32]
vsrc.qword[1] ← VSR[VRB+32]

do i = 0 to 7
   VSR[VRT+32].hword[i] ←
        ui16_CLAMP(EXTZ(vsrc.word[i]))
end
```

Let vsrc be the concatenation of the contents of VSR[VRA+32] followed by the contents of VSR[VRB+32].

For each integer value i from 0 to 7, do the following.

The unsigned integer value in word element i of vsrc is placed into halfword element i of VSR[VRT+32] in unsigned integer format.

- If the value of the element is greater than $2^{16}-1$ the result saturates to $2^{16}-1$ and SAT is set to 1.

Special Registers Altered:

Register	*Field(s)*
VSCR	SAT

Register Data Layout for vpkuwum & vpkuwus

src1	VSR[VRA+32].word[0]	VSR[VRA+32].word[1]	VSR[VRA+32].word[2]	VSR[VRA+32].word[3]
src2	VSR[VRB+32].word[0]	VSR[VRB+32].word[1]	VSR[VRB+32].word[2]	VSR[VRB+32].word[3]

result	VSR[VRT+32].hword[0]	VSR[VRT+32].hword[1]	VSR[VRT+32].hword[2]	VSR[VRT+32].hword[3]	VSR[VRT+32].hword[4]	VSR[VRT+32].hword[5]	VSR[VRT+32].hword[6]	VSR[VRT+32].hword[7]
	0	16	32	48	64	80	96	112 127

Vector Pack Unsigned Doubleword Unsigned Modulo VX-form

vpkudum VRT,VRA,VRB

4	VRT	VRA	VRB	1102
0	6	11	16	21 31

```
if MSR.VEC then Vector_Unavailable()

VSR[VRT+32].word[0] ← VSR[VRA+32].dword[0].bit[32:63]
VSR[VRT+32].word[1] ← VSR[VRA+32].dword[1].bit[32:63]
VSR[VRT+32].word[2] ← VSR[VRB+32].dword[0].bit[32:63]
VSR[VRT+32].word[3] ← VSR[VRB+32].dword[1].bit[32:63]
```

Let vsrc be the concatenation of the contents of VSR[VRA+32] followed by the contents of VSR[VRB+32].

For each integer value i from 0 to 3, do the following.

The contents of bits 32:63 of doubleword element i of vsrc are placed into word element i of VSR[VRT+32].

Special Registers Altered:

None

Vector Pack Unsigned Doubleword Unsigned Saturate VX-form

vpkudus VRT,VRA,VRB

4	VRT	VRA	VRB	1230
0	6	11	16	21 31

```
if MSR.VEC then Vector_Unavailable()

VSR[VRT+32].word[0] ←
        ui32_CLAMP(EXTZ(VSR[VRA+32].dword[0]))
VSR[VRT+32].word[1] ←
        ui32_CLAMP(EXTZ(VSR[VRA+32].dword[1]))
VSR[VRT+32].word[2] ←
        ui32_CLAMP(EXTZ(VSR[VRB+32].dword[0]))
VSR[VRT+32].word[3] ←
        ui32_CLAMP(EXTZ(VSR[VRB+32].dword[1]))
```

Let vsrc be the concatenation of the contents of VSR[VRA+32] followed by the contents of VSR[VRB+32].

For each integer value i from 0 to 3, do the following.

The unsigned integer value in doubleword element i of vsrc are placed into halfword element i of VSR[VRT+32] in unsigned integer format.

- If the value of the element is greater than $2^{32}-1$ the result saturates to $2^{32}-1$ and SAT is set to 1.

Special Registers Altered:

Register	*Field(s)*
VSCR	SAT

Register Data Layout for vpkudum & vpkudus

src1	VSR[VRA+32].dword[0]		VSR[VRA+32].dword[1]	
src2	VSR[VRB+32].dword[0]		VSR[VRB+32].dword[1]	
result	VSR[VRT+32].word[0]	VSR[VRT+32].word[1]	VSR[VRT+32].word[2]	VSR[VRT+32].word[3]
	0	32	64	96 127

6.8.2 Vector Unpack Instructions

Vector Unpack High Signed Byte VX-form

vupkhsb VRT,VRB

4	VRT	///	VRB	526
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 7
   VSR[VRT+32].hword[i] ← EXTS16(VSR[VRB+32].byte[i])
end
```

For each integer value `i` from 0 to 7, do the following.

The signed integer value in byte element `i` of `VSR[VRB+32]` is sign-extended and placed into half-word element `i` in `VSR[VRT+32]`.

Special Registers Altered:

None

Vector Unpack Low Signed Byte VX-form

vupklsb VRT,VRB

4	VRT	///	VRB	654
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 7
   VSR[VRT+32].hword[i] ← EXTS16(VSR[VRB+32].byte[i+8])
end
```

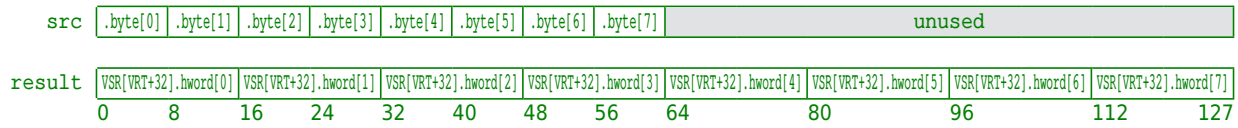
For each integer value `i` from 0 to 7, do the following.

The signed integer value in byte element `i+8` of `VSR[VRB+32]` is sign-extended and placed into half-word element `i` of `VSR[VRT+32]`.

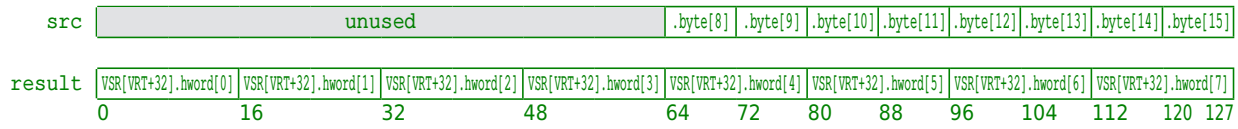
Special Registers Altered:

None

Register Data Layout for vupkhsb



Register Data Layout for vupklsb



Vector Unpack High Signed Halfword VX-form

vupkhsh VRT,VRB

4	VRT	///	VRB	590
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

VSR[VRT+32].word[0] ← EXTS32(VSR[VRB+32].hword[0])
VSR[VRT+32].word[1] ← EXTS32(VSR[VRB+32].hword[1])
VSR[VRT+32].word[2] ← EXTS32(VSR[VRB+32].hword[2])
VSR[VRT+32].word[3] ← EXTS32(VSR[VRB+32].hword[3])
```

For each integer value `i` from 0 to 3, do the following.

The signed integer value in halfword element `i` of `VSR[VRB+32]` is sign-extended and placed into word element `i` in `VSR[VRT+32]`.

Special Registers Altered:

None

Vector Unpack Low Signed Halfword VX-form

vupklsh VRT,VRB

4	VRT	///	VRB	718
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

VSR[VRT+32].word[0] ← EXTS32(VSR[VRB+32].hword[4])
VSR[VRT+32].word[1] ← EXTS32(VSR[VRB+32].hword[5])
VSR[VRT+32].word[2] ← EXTS32(VSR[VRB+32].hword[6])
VSR[VRT+32].word[3] ← EXTS32(VSR[VRB+32].hword[7])
```

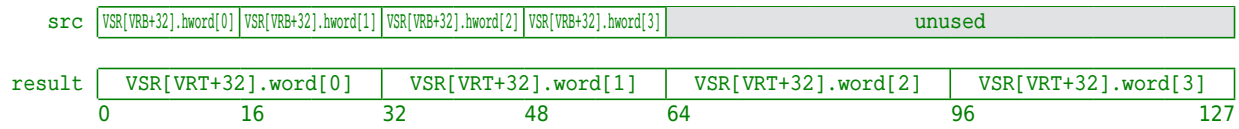
For each integer value `i` from 0 to 3, do the following.

The signed integer value in halfword element `i+4` of `VSR[VRB+32]` is sign-extended to produce a signed-integer word and placed into word element `i` of `VSR[VRT+32]`.

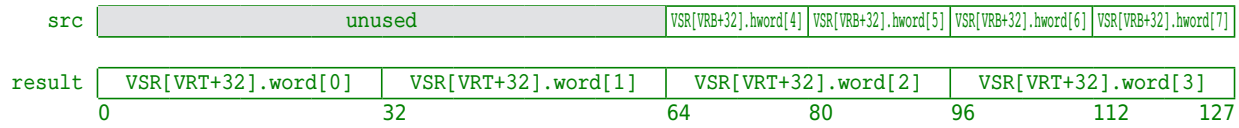
Special Registers Altered:

None

Register Data Layout for vupkhsh



Register Data Layout for vupklsh



Vector Unpack High Signed Word VX-form

vupkhsw VRT,VRB

4	VRT	///	VRB	1614
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

VSR[VRT+32].dword[0] ← EXTS64(VSR[VRB+32].word[0])
VSR[VRT+32].dword[1] ← EXTS64(VSR[VRB+32].word[1])
```

For each integer value `i` from 0 to 1, do the following.

The signed integer value in word element `i` of `VSR[VRB+32]` is sign-extended and placed into doubleword element `i` of `VSR[VRT+32]`.

Special Registers Altered:

None

Vector Unpack Low Signed Word VX-form

vupklsw VRT,VRB

4	VRT	///	VRB	1742
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

VSR[VRT+32].dword[0] ← EXTS64(VSR[VRB+32].word[2])
VSR[VRT+32].dword[1] ← EXTS64(VSR[VRB+32].word[3])
```

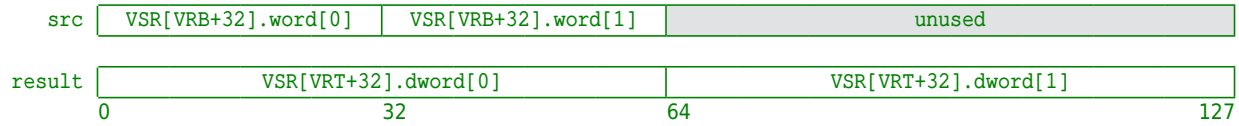
For each integer value `i` from 0 to 1, do the following.

The signed integer value in word element `i+2` of `VSR[VRB+32]` is sign-extended and placed into doubleword element `i` of `VSR[VRT+32]`.

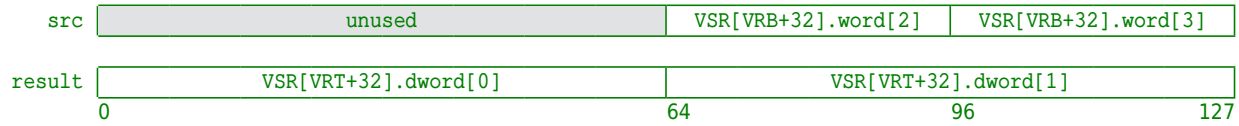
Special Registers Altered:

None

Register Data Layout for vupkhsw



Register Data Layout for vupklsw



Vector Unpack High Pixel VX-form

vupkhpx VRT,VRB

4	VRT	///	VRB	846
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
   src ← VSR[VRB+32].hword[i]

   VSR[VRT+32].word[i].byte[0] ← EXTS8(src.bit[0])
   VSR[VRT+32].word[i].byte[1] ← EXTZ8(src.bit[1:5])
   VSR[VRT+32].word[i].byte[2] ← EXTZ8(src.bit[6:10])
   VSR[VRT+32].word[i].byte[3] ← EXTZ8(src.bit[11:15])
end
```

For each integer value `i` from 0 to 3, do the following.

The contents of halfword element `i` of `VSR[VRB+32]` are unpacked as follows.

- sign-extend bit 0 of the halfword to 8 bits
- zero-extend bits 1:5 of the halfword to 8 bits
- zero-extend bits 6:10 of the halfword to 8 bits
- zero-extend bits 11:15 of the halfword to 8 bits

The result is placed in word element `i` of `VSR[VRT+32]`.

Special Registers Altered:

None

Vector Unpack Low Pixel VX-form

vupklpx VRT,VRB

4	VRT	///	VRB	974
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
   src ← VSR[VRB+32].hword[i+4]

   VSR[VRT+32].word[i].byte[0] ← EXTS8(src.bit[0])
   VSR[VRT+32].word[i].byte[1] ← EXTZ8(src.bit[1:5])
   VSR[VRT+32].word[i].byte[2] ← EXTZ8(src.bit[6:10])
   VSR[VRT+32].word[i].byte[3] ← EXTZ8(src.bit[11:15])
end
```

For each integer value `i` from 0 to 3, do the following.

The contents of halfword element `i+4` of `VSR[VRB+32]` are unpacked as follows.

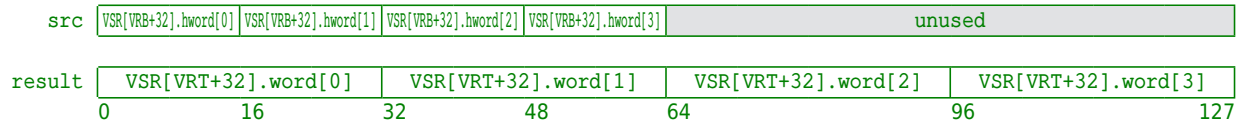
- sign-extend bit 0 of the halfword to 8 bits
- zero-extend bits 1:5 of the halfword to 8 bits
- zero-extend bits 6:10 of the halfword to 8 bits
- zero-extend bits 11:15 of the halfword to 8 bits

The result is placed in word element `i` of `VSR[VRT+32]`.

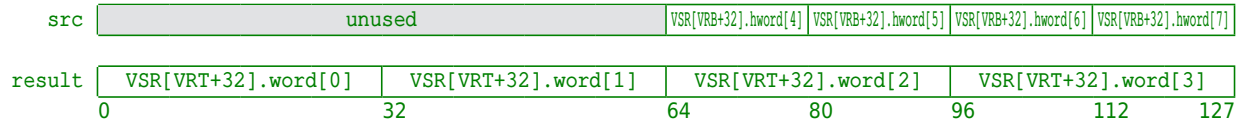
Special Registers Altered:

None

Register Data Layout for vupkhpx



Register Data Layout for vupklpx



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 268.

Programming Note

Notice that the unpacking done by the *Vector Unpack Pixel* instructions does not reverse the packing done by the *Vector Pack Pixel* instruction. Specifically, if a 16-bit pixel is unpacked to a 32-bit pixel which is then packed to a 16-bit pixel, the resulting 16-bit pixel will not, in general, be equal to the original 16-bit pixel (because, for each channel except the first, *Vector Unpack Pixel* inserts high-order bits while *Vector Pack Pixel* discards low-order bits).

6.8.3 Vector Merge Instructions

Vector Merge High Byte VX-form

vmrghb VRT,VRA,VRB

4	VRT	VRA	VRB	12
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 7
   VSR[VRT+32].hword[i].byte[0] ← VSR[VRA+32].byte[i]
   VSR[VRT+32].hword[i].byte[1] ← VSR[VRB+32].byte[i]
end
```

For each integer value `i` from 0 to 7, do the following.

The contents of byte element `i` of `VSR[VRA+32]` are placed into byte element `2×i` of `VSR[VRT+32]`.

The contents of byte element `i` of `VSR[VRB+32]` are placed into byte element `2×i+1` of `VSR[VRT+32]`.

Special Registers Altered:

None

Vector Merge Low Byte VX-form

vmrglb VRT,VRA,VRB

4	VRT	VRA	VRB	268
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 7
   VSR[VRT+32].hword[i].byte[0] ← VSR[VRA+32].byte[i+8]
   VSR[VRT+32].hword[i].byte[1] ← VSR[VRB+32].byte[i+8]
end
```

For each integer value `i` from 0 to 7, do the following.

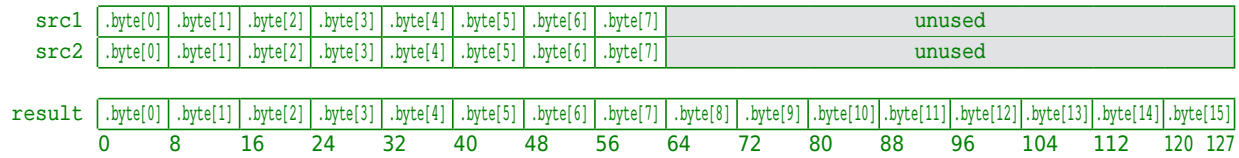
The contents of byte element i+8 of `VSR[VRA+32]` are placed into byte element `2×i` of `VSR[VRT+32]`.

The contents of byte element i+8 of `VSR[VRB+32]` are placed into byte element `2×i+1` of `VSR[VRT+32]`.

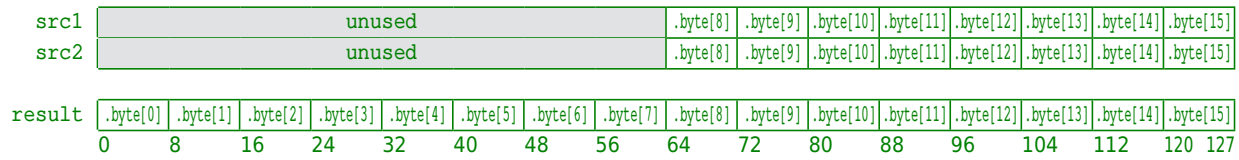
Special Registers Altered:

None

Register Data Layout for vmrghb



Register Data Layout for vmrglb



Vector Merge High Halfword VX-form

vmrghh VRT,VRA,VRB

4	VRT	VRA	VRB	76
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
   VSR[VRT+32].word[i].hword[0] ← VSR[VRA+32].hword[i]
   VSR[VRT+32].word[i].hword[1] ← VSR[VRB+32].hword[i]
end
```

For each integer value `i` from 0 to 3, do the following.

The contents of halfword element `i` of `VSR[VRA+32]` are placed into halfword element `2×i` of `VSR[VRT+32]`.

The contents of halfword element `i` of `VSR[VRB+32]` are placed into halfword element `2×i+1` of `VSR[VRT+32]`.

Special Registers Altered:

None

Vector Merge Low Halfword VX-form

vmrglh VRT,VRA,VRB

4	VRT	VRA	VRB	332
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
   VSR[VRT+32].word[i].hword[0] ← VSR[VRA+32].hword[i+4]
   VSR[VRT+32].word[i].hword[1] ← VSR[VRB+32].hword[i+4]
end
```

For each integer value `i` from 0 to 3, do the following.

The contents of halfword element `i+4` of `VSR[VRA+32]` are placed into halfword element `2×i` of `VSR[VRT+32]`.

The contents of halfword element `i+4` of `VSR[VRB+32]` are placed into halfword element `2×i+1` of `VSR[VRT+32]`.

Special Registers Altered:

None

Register Data Layout for vmrghh

src1	VSR[VRA+32].hword[0]	VSR[VRA+32].hword[1]	VSR[VRA+32].hword[2]	VSR[VRA+32].hword[3]	unused			
src2	VSR[VRB+32].hword[0]	VSR[VRB+32].hword[1]	VSR[VRB+32].hword[2]	VSR[VRB+32].hword[3]	unused			
result	VSR[VRT+32].hword[0]	VSR[VRT+32].hword[1]	VSR[VRT+32].hword[2]	VSR[VRT+32].hword[3]	VSR[VRT+32].hword[4]	VSR[VRT+32].hword[5]	VSR[VRT+32].hword[6]	VSR[VRT+32].hword[7]
	0	16	32	48	64	80	96	112 127

Register Data Layout for vmrglh

src1	unused				VSR[VRA+32].hword[4]	VSR[VRA+32].hword[5]	VSR[VRA+32].hword[6]	VSR[VRA+32].hword[7]
src2	unused				VSR[VRB+32].hword[4]	VSR[VRB+32].hword[5]	VSR[VRB+32].hword[6]	VSR[VRB+32].hword[7]
result	VSR[VRT+32].hword[0]	VSR[VRT+32].hword[1]	VSR[VRT+32].hword[2]	VSR[VRT+32].hword[3]	VSR[VRT+32].hword[4]	VSR[VRT+32].hword[5]	VSR[VRT+32].hword[6]	VSR[VRT+32].hword[7]
	0	16	32	48	64	80	96	112 127

Vector Merge High Word VX-form

vmrghw VRT,VRA,VRB

4	VRT	VRA	VRB	140
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

VSR[VRT+32].word[0] ← VSR[VRA+32].word[0]
VSR[VRT+32].word[1] ← VSR[VRB+32].word[0]
VSR[VRT+32].word[2] ← VSR[VRA+32].word[1]
VSR[VRT+32].word[3] ← VSR[VRB+32].word[1]
```

The contents of word element 0 of `VSR[VRA+32]` are placed into word element 0 of `VSR[VRT+32]`.

The contents of word element 0 of `VSR[VRB+32]` are placed into word element 1 of `VSR[VRT+32]`.

The contents of word element 1 of `VSR[VRA+32]` are placed into word element 2 of `VSR[VRT+32]`.

The contents of word element 1 of `VSR[VRB+32]` are placed into word element 3 of `VSR[VRT+32]`.

Special Registers Altered:

None

Vector Merge Low Word VX-form

vmrglw VRT,VRA,VRB

4	VRT	VRA	VRB	396
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

VSR[VRT+32].word[0] ← VSR[VRA+32].word[2]
VSR[VRT+32].word[1] ← VSR[VRB+32].word[2]
VSR[VRT+32].word[2] ← VSR[VRA+32].word[3]
VSR[VRT+32].word[3] ← VSR[VRB+32].word[3]
```

The contents of word element 2 of `VSR[VRA+32]` are placed into word element 0 of `VSR[VRT+32]`.

The contents of word element 2 of `VSR[VRB+32]` are placed into word element 1 of `VSR[VRT+32]`.

The contents of word element 3 of `VSR[VRA+32]` are placed into word element 2 of `VSR[VRT+32]`.

The contents of word element 3 of `VSR[VRB+32]` are placed into word element 3 of `VSR[VRT+32]`.

Special Registers Altered:

None

Register Data Layout for vmrghw

src1	VSR[VRA+32].word[0]	VSR[VRA+32].word[1]	unused	
src2	VSR[VRB+32].word[0]	VSR[VRB+32].word[1]	unused	
result	VSR[VRT+32].word[0]	VSR[VRT+32].word[1]	VSR[VRT+32].word[2]	VSR[VRT+32].word[3]
	0	32	64	96 127

Register Data Layout for vmrglw

src1	unused		VSR[VRA+32].word[2]	VSR[VRA+32].word[3]
src2	unused		VSR[VRB+32].word[2]	VSR[VRB+32].word[3]
result	VSR[VRT+32].word[0]	VSR[VRT+32].word[1]	VSR[VRT+32].word[2]	VSR[VRT+32].word[3]
	0	32	64	96 127

Vector Merge Even Word VX-form

vmrgew VRT,VRA,VRB

4	VRT	VRA	VRB	1932
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

VSR[VRT+32].word[0] ← VSR[VRA+32].word[0]
VSR[VRT+32].word[1] ← VSR[VRB+32].word[0]
VSR[VRT+32].word[2] ← VSR[VRA+32].word[2]
VSR[VRT+32].word[3] ← VSR[VRB+32].word[2]
```

The contents of word element 0 of `VSR[VRA+32]` are placed into word element 0 of `VSR[VRT+32]`.

The contents of word element 0 of `VSR[VRB+32]` are placed into word element 1 of `VSR[VRT+32]`.

The contents of word element 2 of `VSR[VRA+32]` are placed into word element 2 of `VSR[VRT+32]`.

The contents of word element 2 of `VSR[VRB+32]` are placed into word element 3 of `VSR[VRT+32]`.

vmrgew is treated as a *Vector* instruction in terms of resource availability.

Special Registers Altered:

None

Vector Merge Odd Word VX-form

vmrgow VRT,VRA,VRB

4	VRT	VRA	VRB	1676
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

VSR[VRT+32].word[0] ← VSR[VRA+32].word[1]
VSR[VRT+32].word[1] ← VSR[VRB+32].word[1]
VSR[VRT+32].word[2] ← VSR[VRA+32].word[3]
VSR[VRT+32].word[3] ← VSR[VRB+32].word[3]
```

The contents of word element 1 of `VSR[VRA+32]` are placed into word element 0 of `VSR[VRT+32]`.

The contents of word element 1 of `VSR[VRB+32]` are placed into word element 1 of `VSR[VRT+32]`.

The contents of word element 3 of `VSR[VRA+32]` are placed into word element 2 of `VSR[VRT+32]`.

The contents of word element 3 of `VSR[VRB+32]` are placed into word element 3 of `VSR[VRT+32]`.

vmrgow is treated as a *Vector* instruction in terms of resource availability.

Special Registers Altered:

None

Register Data Layout for vmrgew

src1	VSR[VRA+32].word[0]	unused	VSR[VRA+32].word[2]	unused
src2	VSR[VRB+32].word[0]	unused	VSR[VRB+32].word[2]	unused
result	VSR[VRT+32].word[0]	VSR[VRT+32].word[1]	VSR[VRT+32].word[2]	VSR[VRT+32].word[3]
	0	32	64	96 127

Register Data Layout for vmrgow

src1	unused	VSR[VRA+32].word[1]	unused	VSR[VRA+32].word[3]
src2	unused	VSR[VRB+32].word[1]	unused	VSR[VRB+32].word[3]
result	VSR[VRT+32].word[0]	VSR[VRT+32].word[1]	VSR[VRT+32].word[2]	VSR[VRT+32].word[3]
	0	32	64	96 127

6.8.4 Vector Splat Instructions

Programming Note

The *Vector Splat* instructions can be used in preparation for performing arithmetic for which one source vector is to consist of elements that all have the same value (e.g., multiplying all elements of a VSR by a constant).

Vector Splat Byte VX-form

vspltb VRT,VRB,UIM

4	VRT	/	UIM	VRB	524
0	6	11	12	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

b ← UIM || 0b000
do i = 0 to 15
   VSR[VRT+32].byte[i] ← VSR[VRB+32].bit[b:b+7]
end
```

For each integer value `i` from 0 to 15, do the following.

The contents of byte element `UIM` in `VSR[VRB+32]` are placed into byte element `i` of `VSR[VRT+32]`.

Special Registers Altered:

None

Vector Splat Halfword VX-form

vsplth VRT,VRB,UIM

4	VRT	//	UIM	VRB	588
0	6	11	13	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

b ← UIM || 0b0000
do i = 0 to 7
   VSR[VRT+32].hword[i] ← VSR[VRB+32].bit[b:b+15]
end
```

For each integer value `i` from 0 to 7, do the following.

The contents of halfword element `UIM` in `VSR[VRB+32]` are placed into halfword element `i` of `VSR[VRT+32]`.

Special Registers Altered:

None

Register Data Layout for vspltb

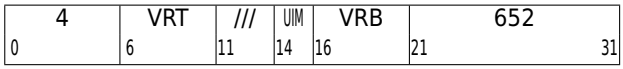
src	.byte[0]	.byte[1]	.byte[2]	.byte[3]	.byte[4]	.byte[5]	.byte[6]	.byte[7]	.byte[8]	.byte[9]	.byte[10]	.byte[11]	.byte[12]	.byte[13]	.byte[14]	.byte[15]
result	.byte[0]	.byte[1]	.byte[2]	.byte[3]	.byte[4]	.byte[5]	.byte[6]	.byte[7]	.byte[8]	.byte[9]	.byte[10]	.byte[11]	.byte[12]	.byte[13]	.byte[14]	.byte[15]
	0	8	16	24	32	40	48	56	64	72	80	88	96	104	112	120 127

Register Data Layout for vsplth

src	VSR[VRB+32].hword[0]	VSR[VRB+32].hword[1]	VSR[VRB+32].hword[2]	VSR[VRB+32].hword[3]	VSR[VRB+32].hword[4]	VSR[VRB+32].hword[5]	VSR[VRB+32].hword[6]	VSR[VRB+32].hword[7]
result	VSR[VRT+32].hword[0]	VSR[VRT+32].hword[1]	VSR[VRT+32].hword[2]	VSR[VRT+32].hword[3]	VSR[VRT+32].hword[4]	VSR[VRT+32].hword[5]	VSR[VRT+32].hword[6]	VSR[VRT+32].hword[7]
	0	16	32	48	64	80	96	112 127

Vector Splat Word VX-form

vspltw VRT,VRB,UIM




```
if MSR.VEC=0 then Vector_Unavailable()

b ← UIM || 0b00000
do i = 0 to 3
   VSR[VRT+32].word[i] ← VSR[VRB+32].bit[b:b+31]
end
```

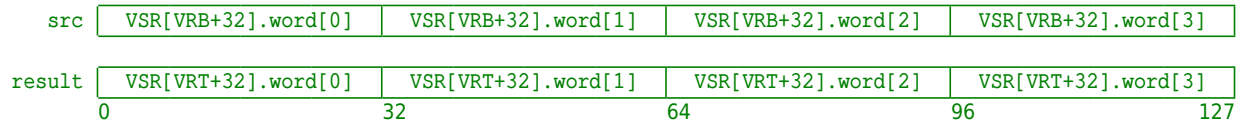
For each integer value `i` from 0 to 3, do the following.

The contents of word element `UIM` in `VSR[VRB+32]` are placed into word element `i` of `VSR[VRT+32]`.

Special Registers Altered:

None

Register Data Layout for vspltw



Vector Splat Immediate Signed Byte VX-form

vspltisb VRT,SIM

4	VRT	SIM	///	780
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 15
   VSR[VRT+32].byte[i] ← EXTS8(SIM, 8)
end
```

For each integer value `i` from 0 to 15, do the following.

The value of the `SIM` field, sign-extended to 8 bits, is placed into byte element `i` of `VSR[VRT+32]`.

Special Registers Altered:

None

Vector Splat Immediate Signed Halfword VX-form

vspltish VRT,SIM

4	VRT	SIM	///	844
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 7
   VSR[VRT+32].hword[i] ← EXTS16(SIM, 16)
end
```

For each integer value `i` from 0 to 7, do the following.

The value of the `SIM` field, sign-extended to 16 bits, is placed into halfword element `i` of `VSR[VRT+32]`.

Special Registers Altered:

None

Vector Splat Immediate Signed Word VX-form

vspltisw VRT,SIM

4	VRT	SIM	///	908
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
   VSR[VRT+32].word[i] ← EXTS32(SIM, 32)
end
```

For each integer value `i` from 0 to 3, do the following.

The value of the `SIM` field, sign-extended to 32 bits, is placed into word element `i` of `VSR[VRT+32]`.

Special Registers Altered:

None

Register Data Layout for vspltisb

result	.byte[0]	.byte[1]	.byte[2]	.byte[3]	.byte[4]	.byte[5]	.byte[6]	.byte[7]	.byte[8]	.byte[9]	.byte[10]	.byte[11]	.byte[12]	.byte[13]	.byte[14]	.byte[15]
	0	8	16	24	32	40	48	56	64	72	80	88	96	104	112	120 127

Register Data Layout for vspltish

result	VSR[VRT+32].hword[0]	VSR[VRT+32].hword[1]	VSR[VRT+32].hword[2]	VSR[VRT+32].hword[3]	VSR[VRT+32].hword[4]	VSR[VRT+32].hword[5]	VSR[VRT+32].hword[6]	VSR[VRT+32].hword[7]
	0	16	32	48	64	80	96	112 127

Register Data Layout for vspltisw

result	VSR[VRT+32].word[0]	VSR[VRT+32].word[1]	VSR[VRT+32].word[2]	VSR[VRT+32].word[3]
	0	32	64	96 127

6.8.5 Vector Permute Instruction

The *Vector Permute* instruction allows any byte in two source VSRs to be copied to any byte in the target VSR. The bytes in a third source VSR specify from which byte in the first two source VSRs the corresponding target byte is to be copied. The contents of the third source VSR are sometimes referred to as the “permute control vector”.

Vector Permute VA-form

vperm VRT,VRA,VRB,VRC

4	VRT	VRA	VRB	VRC	43
0	6	11	16	21	26 31

```
if MSR.VEC=0 then Vector_Unavailable()

vsrc.qword[0] ← VSR[VRA+32]
vsrc.qword[1] ← VSR[VRB+32]

do i = 0 to 15
   index ← VSR[VRC+32].byte[i].bit[3:7]
   VSR[VRT+32].byte[i] ← src.byte[index]
end
```

Let the source vector be the concatenation of the contents of VSR[VRA+32] followed by the contents of VSR[VRB+32].

For each integer value i from 0 to 15, do the following.

Let index be the value specified by bits 3:7 of byte element i of VSR[VRC+32].

The contents of byte element index of src are placed into byte element i of VSR[VRT+32].

Special Registers Altered:

None

Programming Note

See the Programming Notes with the *Load Vector for Shift Left* and *Load Vector for Shift Right* instructions on page 266 for examples of uses of ***vperm***.

Vector Permute Right-indexed VA-form

vpermr VRT,VRA,VRB,VRC

4	VRT	VRA	VRB	VRC	59
0	6	11	16	21	26 31

```
if MSR.VEC=0 then Vector_Unavailable()

vsrc.qword[0] ← VSR[VRA+32]
vsrc.qword[1] ← VSR[VRB+32]

do i = 0 to 15
   index ← VSR[VRC+32].byte[i].bit[3:7]
   VSR[VRT+32].byte[i] ← src.byte[31-index]
end
```

Let the source vector be the concatenation of the contents of VSR[VRA+32] followed by the contents of VSR[VRB+32].

For each integer value i from 0 to 15, do the following.

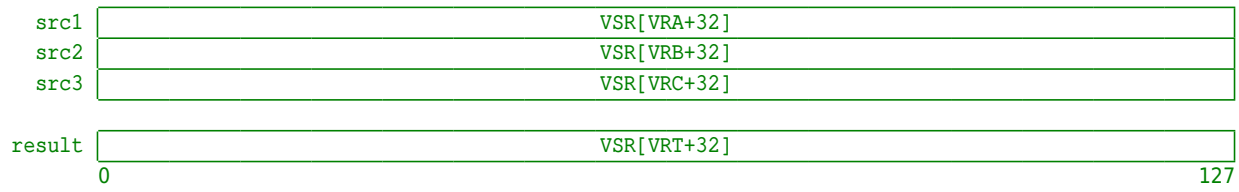
Let index be the value specified by bits 3:7 of byte element i of VSR[VRC+32].

The contents of byte element 31-index of src are placed into byte element i of VSR[VRT+32].

Special Registers Altered:

None

Register Data Layout for vperm & vpermr



6.8.6 Vector Select Instruction

Vector Select VA-form

vsel VRT,VRA,VRB,VRC

4	VRT	VRA	VRB	VRC	42
0	6	11	16	21	26 31

```
if MSR.VEC=0 then Vector_Unavailable()

src1 ← VSR[VRA+32]
src2 ← VSR[VRB+32]
mask ← VSR[VRC+32]

VSR[VRT+32] ← (src1 & ¬mask) | (src2 & mask)
```

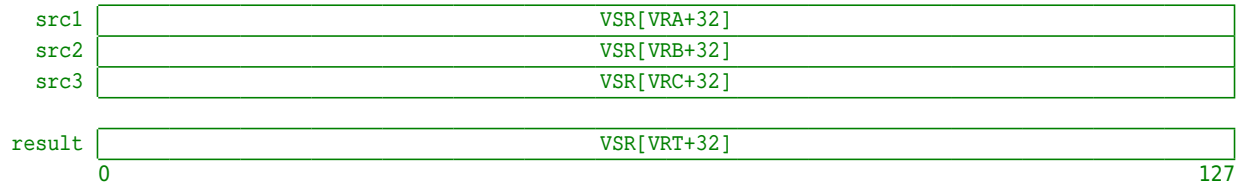
Let `src1` be the contents of `VSR[VRA+32]`.
Let `src2` be the contents of `VSR[VRB+32]`.
Let `mask` be the contents of `VSR[VRC+32]`.

The value, `(src1 & ¬mask) | (src2 & mask)`, is placed into `VSR[VRT+32]`.

Special Registers Altered:

None

Register Data Layout for vsel



6.8.7 Vector Shift Instructions

The *Vector Shift* instructions rotate or shift the contents of a VSR or a pair of VSRs left or right by a specified number of bytes (***vslo***, ***vsro***, ***vsldoi***) or bits (***vsl***, ***vsr***). Depending on the instruction, this "shift count" is specified either by the contents of a VSR or by an immediate field in the instruction. In the former case, 7 bits of the shift count register give the shift count in bits ($0 \leq$ count $\leq$ 127). Of these 7 bits, the high-order 4 bits give the number of complete bytes by which to shift and are used by ***vslo*** and ***vsro***; the low-order 3 bits give the number of remaining bits by which to shift and are used by ***vsl*** and ***vsr***.

Programming Note

A pair of these instructions, specifying the same shift count register, can be used to shift the contents of a VSR left or right by the number of bits (0-127) specified in the shift count register. The following example shifts the contents of register `Vx` left by the number of bits specified in register `Vy` and places the result into register `Vz`.

```
vslo      Vz,Vx,Vy
vspltb    Vy,Vy,15
vsl       Vz,Vz,Vy
```

Vector Shift Left Double by Bit Immediate VN-form

vsldbi VRT,VRA,VRB,SH

4	VRT	VRA	VRB	0	SH	22
0	6	11	16	21	23	26 31

```
if MSR.VEC=0 then Vector_Unavailable()

vsrc.qword[0] ← VSR[VRA+32]
vsrc.qword[1] ← VSR[VRB+32]

VSR[VRT+32]   ← vsrc.bit[SH:SH+127]
```

Let vsrc be the contents of VSR[VRA+32] concatenated with the contents of VSR[VRB+32].

The contents of bits SH:SH+127 of vsrc are placed into VSR[VRT+32].

SH can be any integer value between 0 and 7.

Special Registers Altered:

None

Vector Shift Left Double by Octet Immediate VA-form

vsldoi VRT,VRA,VRB,SHB

4	VRT	VRA	VRB	/	SHB	44
0	6	11	16	21	22	26 31

```
if MSR.VEC=0 then Vector_Unavailable()

vsrc.qword[0] ← VSR[VRA+32]
vsrc.qword[1] ← VSR[VRB+32]

VSR[VRT+32] ← src.byte[SHB:SHB+15]
```

Let vsrc be the contents of VSR[VRA+32] concatenated with the contents of VSR[VRB+32].

Bytes SHB:SHB+15 of vsrc are placed into VSR[VRT+32].

Special Registers Altered:

None

Vector Shift Right Double by Bit Immediate VN-form

vsrdbi VRT,VRA,VRB,SH

4	VRT	VRA	VRB	1	SH	22
0	6	11	16	21	23	26 31

```
if MSR.VEC=0 then Vector_Unavailable()

vsrc.qword[0] ← VSR[VRA+32]
vsrc.qword[1] ← VSR[VRB+32]

VSR[VRT+32]   ← vsrc.bit[128-SH:255-SH]
```

Let vsrc be the contents of VSR[VRA+32] concatenated with the contents of VSR[VRB+32].

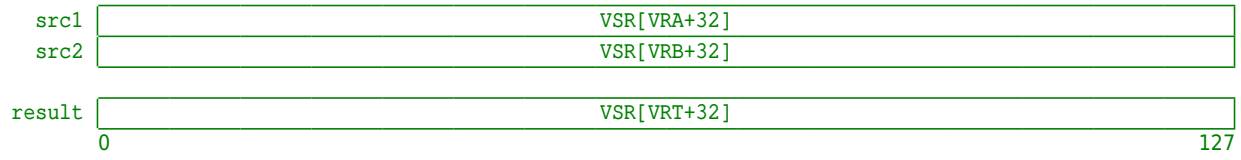
The contents of bits 128-SH:255-SH of vsrc are placed into VSR[VRT+32].

SH can be any integer value between 0 and 7.

Special Registers Altered:

None

Register Data Layout for vsldbi, & vsrdbi & vsldoi



Vector Shift Left VX-form

vsl VRT,VRA,VRB

4	VRT	VRA	VRB	452
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

sh ← VSR[VRB+32].bit[125:127]

t ← 1
do i = 0 to 14
   t ← t & (VSR[VRB+32].byte[i].bit[5:7] = sh)
end
if t=1 then
   VSR[VRT+32] ← VSR[VRA+32] << sh
else
   VSR[VRT+32] ← UNDEFINED
```

The contents of VSR[VRA+32] are shifted left by the number of bits specified in bits 125:127 of VSR[VRB+32].

- Bits shifted out of bit 0 are lost.
- Zeros are supplied to the vacated bits on the right.

The result is placed into VSR[VRT+32], except if, for any byte element in VSR[VRB+32], the low-order 3 bits are not equal to the shift amount, then VSR[VRT+32] is undefined.

Special Registers Altered:

None

Vector Shift Right VX-form

vsr VRT,VRA,VRB

4	VRT	VRA	VRB	708
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

sh ← VSR[VRB+32].bit[125:127]

t ← 1
do i = 0 to 14
   t ← t & (VSR[VRB+32].byte[i].bit[5:7] = sh)
end
if t=1 then
   VSR[VRT+32] ← CHOP128(EXTZ(VSR[VRA+32]) >> sh)
else
   VSR[VRT+32] ← UNDEFINED
```

The contents of VSR[VRA+32] are shifted right by the number of bits specified in bits 125:127 of VSR[VRB+32].

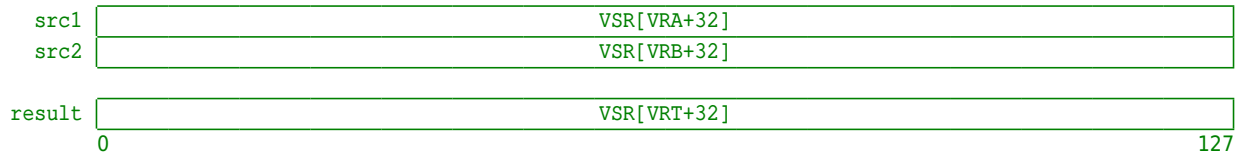
- Bits shifted out of bit 127 are lost.
- Zeros are supplied to the vacated bits on the left.

The result is place into VSR[VRT+32], except if, for any byte element in VSR[VRB+32], the low-order 3 bits are not equal to the shift amount, then VSR[VRT+32] is undefined.

Special Registers Altered:

None

Register Data Layout for vsl & vsr



Vector Shift Left by Octet VX-form

vslo VRT,VRA,VRB

4	VRT	VRA	VRB	1036
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

shb ← VSR[VRB+32].bit[121:124] << 3

VSR[VRT+32] ← VSR[VRA+32] << shb
```

The contents of VSR[VRA+32] are shifted left by the number of bytes specified in bits 121:124 of VSR[VRB+32].

- Bytes shifted out of byte 0 are lost.
- Zeros are supplied to the vacated bytes on the right.

The result is placed into VSR[VRT+32].

Special Registers Altered:

None

Vector Shift Right by Octet VX-form

vsro VRT,VRA,VRB

4	VRT	VRA	VRB	1100
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

shb ← VSR[VRB+32].bit[121:124] << 3

VSR[VRT+32] ← VSR[VRA+32] >> shb
```

The contents of VSR[VRA+32] are shifted right by the number of bytes specified in bits 121:124 of VSR[VRB+32].

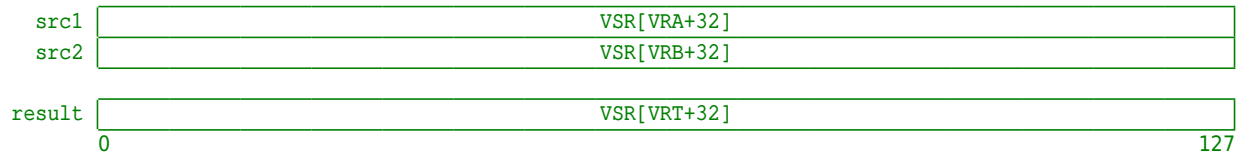
- Bytes shifted out of byte 15 are lost.
- Zeros are supplied to the vacated bytes on the left.

The result is placed into VSR[VRT+32].

Special Registers Altered:

None

Register Data Layout for vslo & vsro



Programming Note

A double-register shift by a dynamically specified number of bits (0-127) can be performed in six instructions. The following example shifts Vw || Vx left by the number of bits specified in Vy and places the high-order 128 bits of the result into Vz.

```
vslo        Vt1,Vw,Vy      # shift high-order reg left
vspltb      Vy,Vy,15
vsl         Vt1,Vt1,Vy
vsububm     Vt3,V0,Vy      # adjust shift count ((V0)=0)
vsro        Vt2,Vx,Vt3     # shift low-order reg right
vspltb      Vt3,Vt3,15
vsr         Vt2,Vt2,Vt3
vor         Vz,Vt1,Vt2     # merge to get final result
```

Vector Shift Left Variable VX-form

vslv VRT,VRA,VRB

4	VRT	VRA	VRB	1860
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable_Interrupt()

vsrc.byte[0:15] ← VSR[VRA+32]
vsrc.byte[16]   ← 0x00

do i = 0 to 15
   sh ← VSR[VRB+32].byte[i].bit[5:7]
   VSR[VRT+32].byte[i] ← src.byte[i:i+1].bit[sh:sh+7]
end
```

Let bytes 0:15 of `vsrc` be the contents of `VSR[VRA+32]`.

Let byte 16 of `vsrc` 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 `VSR[VRB+32]`.
>
> The contents of bits `sh:sh+7` of the halfword in byte elements `i:i+1` of `vsrc` are placed into byte element `i` of `VSR[VRT+32]`.

Special Registers Altered:

None

Vector Shift Right Variable VX-form

vsrv VRT,VRA,VRB

4	VRT	VRA	VRB	1796
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable_Interrupt()

vsrc.byte[0]   ← 0x00
vsrc.byte[1:16] ← VSR[VRA+32]

do i = 0 to 15
   sh ← VSR[VRB+32].byte[i].bit[5:7]
   VSR[VRT+32].byte[i] ← src.byte[i:i+1].bit[8-sh:15-sh]
end
```

Let bytes 1:16 of `vsrc` be the contents of `VSR[VRA+32]`.

Let byte 0 of `vsrc` be the value `0x00`.

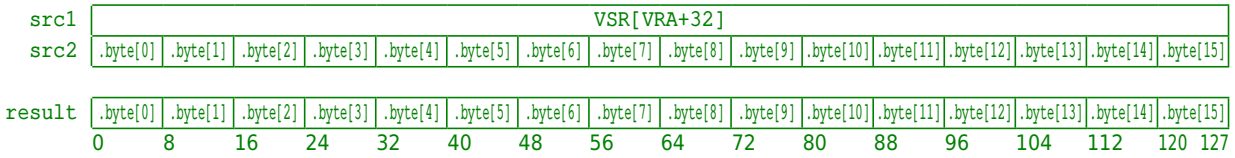
For each integer value i from 0 to 15, do the following.

> Let `sh` be the value in bits 5:7 of byte element `i` of `VSR[VRB+32]`.
>
> The contents of bits `8-sh:15-sh` of the halfword in byte elements `i:i+1` of `vsrc` are placed into byte element `i` of `VSR[VRT+32]`.

Special Registers Altered:

None

Register Data Layout for vslv & vsrv



Programming Note

Assume vSRC contains a vector of packed 7-bit values, A located in bits 0:6, B located in bits 7:13, C located in bits 14:20, etc..

```
                                # vSRC    = { 0bAAAAAAAB, 0bBBBBBBCC, 0bCCCCCDDD, 0bDDDDEEEE,
                                #             0bEEEFFFFF, 0bFFGGGGGG, 0bGHHHHHHH, 0bIIIIIIIJ,
                                #             0bJJJJJJKK, 0bKKKKKLLL, 0bLLLLMMMM, 0bMMMNNNNN,
                                #             0bNNOOOOOO, 0bOPPPPPPP, 0bQQQQQQQR, 0bRRRRRRSS };
```

Assume the following registers are pre-loaded as follows,

```
                                # vSHCNT1 = { 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x07,
                                #             0x07, 0x07, 0x07, 0x07, 0x07, 0x07, 0x07, 0x07 };
                                # vSHCNT2 = { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x01,
                                #             0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x07, 0x07 };
                                # vSHCNT3 = { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
                                #             0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x01, 0x02 };
                                # vMASK   = { 0x7F, 0x7F, 0x7F, 0x7F, 0x7F, 0x7F, 0x7F, 0x7F,
                                #             0x7F, 0x7F, 0x7F, 0x7F, 0x7F, 0x7F, 0x7F, 0x7F };
```

The leftmost seven packed 7-bit values can be unpacked into byte elements 0 to 6 using *vsrv* with vSHCNT1.

```
vsrv    vTMP1, vSRC, vSHCT1   # vTMP1   = { 0b0AAAAAAA, 0bABBBBBBB, 0bBCCCCCCC, 0bCDDDDDDD,
                              #             0bDEEEEEEE, 0bEFFFFFFF, 0bFGGGGGGG, 0bHHHHHHHI,
                              #             0bIIIIIIJJ, 0bJJJJJKKK, 0bKKKKLLLL, 0bLLLMMMMM,
                              #             0bMMNNNNNN, 0bNOOOOOOO, 0bPPPPPPPQ, 0bQQQQQQRR };
```

The next seven packed 7-bit values can then be unpacked into byte elements 7 to 13 using *vsrv* with vSHCNT2.

```
vsrv    vTMP2, vTMP1, vSHCT2 # vTMP2   = { 0b0AAAAAAA, 0bABBBBBBB, 0bBCCCCCCC, 0bCDDDDDDD,
                             #             0bDEEEEEEE, 0bEFFFFFFF, 0bFGGGGGGG, 0bGHHHHHHH,
                             #             0bHIIIIIII, 0bIJJJJJJJ, 0bJKKKKKKK, 0bKLLLLLLL,
                             #             0bLMMMMMMM, 0bMNNNNNNN, 0bOOOOOOOP, 0bPPPPPPQQ };
```

The next two packed 7-bit values can then be unpacked into byte elements 14 to 15 using *vsrv* with vSHCNT3.

```
vsrv    vTMP3, vTMP2, vSHCT3 # vTMP3   = { 0b0AAAAAAA, 0bABBBBBBB, 0bBCCCCCCC, 0bCDDDDDDD,
                             #             0bDEEEEEEE, 0bEFFFFFFF, 0bFGGGGGGG, 0bGHHHHHHH,
                             #             0bHIIIIIII, 0bIJJJJJJJ, 0bJKKKKKKK, 0bKLLLLLLL,
                             #             0bLMMMMMMM, 0bMNNNNNNN, 0bNOOOOOOO, 0bOPPPPPPP };
```

The most-significant bit in each byte element is masked off to produce a vector of sixteen unsigned byte elements.

```
vand    vTMP4, vTMP3, vMASK  # vTMP4   = { 0b0AAAAAAA, 0b0BBBBBBB, 0b0CCCCCCC, 0b0DDDDDDD,
                             #             0b0EEEEEEE, 0b0FFFFFFF, 0b0GGGGGGG, 0b0HHHHHHH,
                             #             0b0IIIIIII, 0b0JJJJJJJ, 0b0KKKKKKK, 0b0LLLLLLL,
                             #             0b0MMMMMMM, 0b0NNNNNNN, 0b0OOOOOOO, 0b0PPPPPPP };
```

The vector of sixteen unsigned byte elements can be further unpacked to two vectors of eight unsigned halfword elements using a *vupkhsb* and a *vupklsb*.

```
vupkhsb vTMP5, vTMP4         # vTMP5 = { 0b00000000_0AAAAAAA, 0b00000000_0BBBBBBB, ... };
vupklsb vTMP6, vTMP4         # vTMP6 = { 0b00000000_0IIIIIII, 0b00000000_0JJJJJJJ, ... };
```

The resultant two vectors of eight unsigned halfword elements can then be further unpacked to four vectors of four unsigned word elements using two *vupkhsh* and two *vupklsh* instructions.

```
vupkhsh vRESULT0, vTMP5      # vRESULT0 = { 0b00000000_00000000_00000000_0AAAAAAA, ... };
vupklsh vRESULT1, vTMP5      # vRESULT1 = { 0b00000000_00000000_00000000_0EEEEEEE, ... };
vupkhsh vRESULT2, vTMP6      # vRESULT2 = { 0b00000000_00000000_00000000_0IIIIIII, ... };
vupklsh vRESULT3, vTMP6      # vRESULT3 = { 0b00000000_00000000_00000000_0MMMMMMM, ... };
```

6.8.8 Vector Extract Element Instructions

6.8.8.1 Vector Extract Element to VSR using Immediate-specified Index Instructions

Vector Extract Unsigned Byte to VSR using immediate-specified index VX-form

vextractub VRT,VRB,UIM

4	VRT	/	UIM	VRB	525
0	6	11	12	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

src ← VSR[VRB+32].byte[UIM]

VSR[VRT+32].dword[0] ← EXTZ64(src)
VSR[VRT+32].dword[1] ← 0x0000_0000_0000_0000
```

The contents of byte element UIM of VSR[VRB+32] are placed into bits 56:63 of VSR[VRT+32]. The contents of the remaining byte elements of VSR[VRT+32] are set to 0.

Special Registers Altered:

None

Vector Extract Unsigned Halfword to VSR using immediate-specified index VX-form

vextractuh VRT,VRB,UIM

4	VRT	/	UIM	VRB	589
0	6	11	12	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

src ← VSR[VRB+32].byte[UIM:UIM+1]

VSR[VRT+32].dword[0] ← EXTZ64(src)
VSR[VRT+32].dword[1] ← 0x0000_0000_0000_0000
```

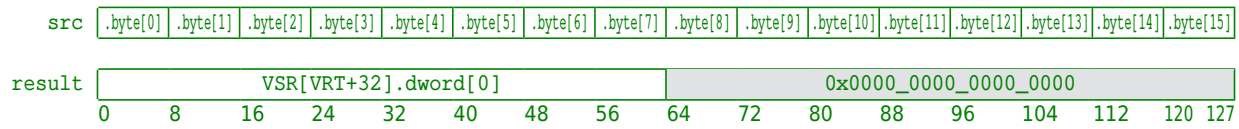
The contents of byte elements UIM:UIM+1 of VSR[VRB+32] are placed into halfword element 3 of VSR[VRT+32]. The contents of the remaining halfword elements of VSR[VRT+32] are set to 0.

If the value of UIM is greater than 14, the results are undefined.

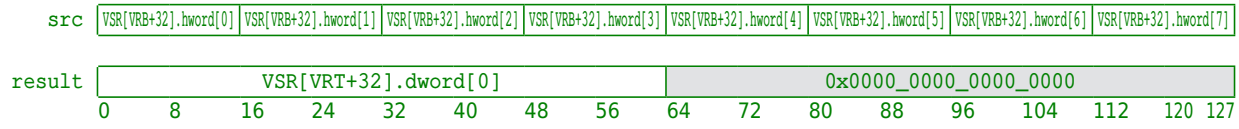
Special Registers Altered:

None

Register Data Layout for vextractub



Register Data Layout for vextractuh



Vector Extract Unsigned Word to VSR using immediate-specified index VX-form

vextractuw VRT,VRB,UIM

4	VRT	/	UIM	VRB	653
0	6	11	12	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

src ← VSR[VRB+32].byte[UIM:UIM+3]

VSR[VRT+32].dword[0] ← EXTZ64(src)
VSR[VRT+32].dword[1] ← 0x0000_0000_0000_0000
```

The contents of byte elements `UIM:UIM+3` of `VSR[VRB+32]` are placed into word element 1 of `VSR[VRT+32]`. The contents of the remaining word elements of `VSR[VRT+32]` are set to 0.

If the value of `UIM` is greater than 12, the results are undefined.

Special Registers Altered:

None

Vector Extract Doubleword to VSR using immediate-specified index VX-form

vextractd VRT,VRB,UIM

4	VRT	/	UIM	VRB	717
0	6	11	12	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

src ← VSR[VRB+32].byte[UIM:UIM+7]

VSR[VRT+32].dword[0] ← src
VSR[VRT+32].dword[1] ← 0x0000_0000_0000_0000
```

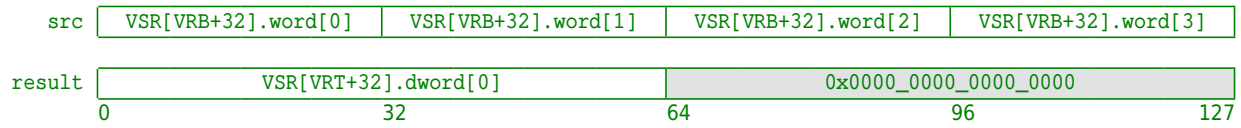
The contents of byte elements `UIM:UIM+7` of `VSR[VRB+32]` are placed into `VSR[VRT+32]`. The contents of doubleword element 1 of `VSR[VRT+32]` are set to 0.

If the value of `UIM` is greater than 8, the results are undefined.

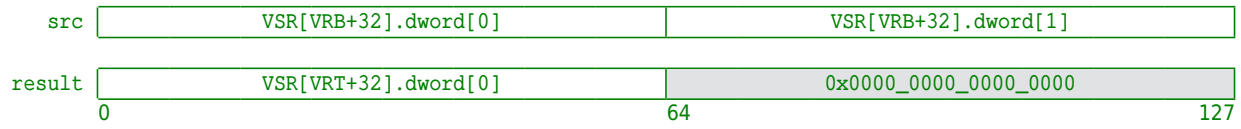
Special Registers Altered:

None

Register Data Layout for vextractuw



Register Data Layout for vextractd



6.8.8.2 Vector Extract Element to GPR using GPR-specified Index Instructions

Vector Extract Unsigned Byte to GPR using GPR-specified Left-Index VX-form

vextublx RT,RA,VRB

4	RT	RA	VRB	1549
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

index ← GPR[RA].bit[60:63]
GPR[RT] ← EXTZ64(VSR[VRB+32].byte[index])
```

Let index be the contents of bits 60:63 of GPR[RA].

The contents of byte element index of VSR[VRB+32] are placed into bits 56:63 of GPR[RT].

The contents of bits 0:55 of GPR[RT] are set to 0.

Special Registers Altered:

None

Vector Extract Unsigned Byte to GPR using GPR-specified Right-Index VX-form

vextubrx RT,RA,VRB

4	RT	RA	VRB	1805
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

index ← GPR[RA].bit[60:63]
GPR[RT] ← EXTZ64(VSR[VRB+32].byte[15-index])
```

Let index be the contents of bits 60:63 of GPR[RA].

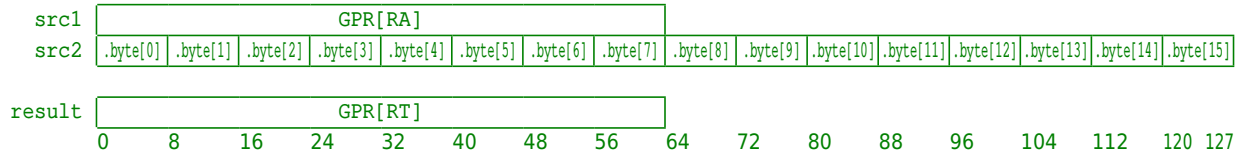
The contents of byte element 15-index of VSR[VRB+32] are placed into bits 56:63 of GPR[RT].

The contents of bits 0:55 of GPR[RT] are set to 0.

Special Registers Altered:

None

Register Data Layout for vextublx & vextubrx



Vector Extract Unsigned Halfword to GPR using GPR-specified Left-Index VX-form

vextuhlx RT,RA,VRB

4	RT	RA	VRB	1613
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

index ← GPR[RA].bit[60:63]
GPR[RT] ← EXTZ64(VSR[VRB+32].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 VSR[VRB+32] are placed into bits 48:63 of GPR[RT].

The contents of bits 0:47 of GPR[RT] are set to 0.

If the value of index is greater than 14, the results are undefined.

Special Registers Altered:

None

Vector Extract Unsigned Halfword to GPR using GPR-specified Right-Index VX-form

vextuhrx RT,RA,VRB

4	RT	RA	VRB	1869
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

index ← GPR[RA].bit[60:63]
GPR[RT] ← EXTZ64(VSR[VRB+32].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 VSR[VRB+32] are placed into bits 48:63 of GPR[RT].

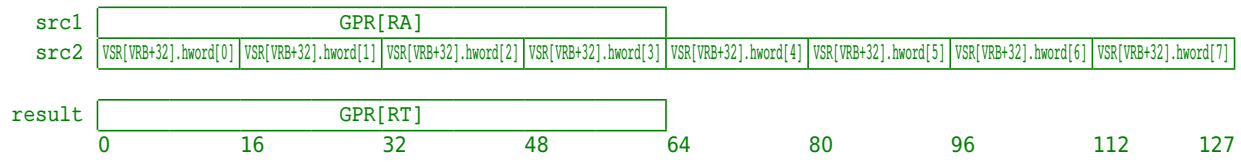
The contents of bits 0:47 of GPR[RT] are set to 0.

If the value of index is greater than 14, the results are undefined.

Special Registers Altered:

None

Register Data Layout for vextuhlx & vextuhrx



Vector Extract Unsigned Word to GPR using GPR-specified Left-Index VX-form

vextuwlx RT,RA,VRB

4	RT	RA	VRB	1677
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

index ← GPR[RA].bit[60:63]
GPR[RT] ← EXTZ64(VSR[VRB+32].byte[index:index+3])
```

Let index be the contents of bits 60:63 of GPR[RA].

The contents of byte elements index:index+3 of VSR[VRB+32] 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 to GPR using GPR-specified Right-Index VX-form

vextuwrx RT,RA,VRB

4	RT	RA	VRB	1933
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

index ← GPR[RA].bit[60:63]
GPR[RT] ← EXTZ64(VSR[VRB+32].byte[12-index:15-index])
```

Let index be the contents of bits 60:63 of GPR[RA].

The contents of byte elements index:index+3 of VSR[VRB+32] are placed into bits 32:63 of GPR[RT].

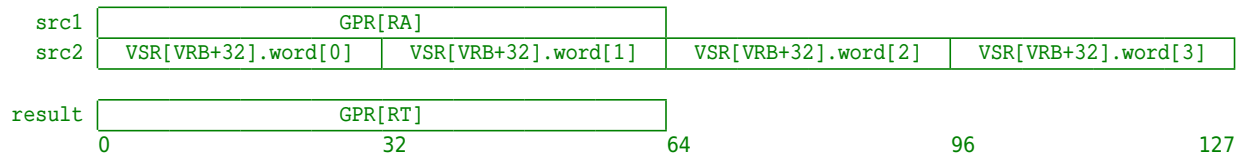
The contents of bits 0:31 of GPR[RT] are set to 0.

If the value of index is greater than 12, the results are undefined.

Special Registers Altered:

None

Register Data Layout for vextuwlx & vextuwrx



6.8.8.3 Vector Extract Double Element to VSR Using GPR-specified Index Instructions

Vector Extract Double Unsigned Byte to VSR using GPR-specified Left-Index VA-form

vextdubvlx VRT,VRA,VRB,RC

4	VRT	VRA	VRB	RC	24
0	6	11	16	21	26 31

```
if MSR.VEC=0 then Vector_Unavailable()

index ← GPR[RC].bit[59:63]
vsrc.qword[0] ← VSR[VRA+32]
vsrc.qword[1] ← VSR[VRB+32]

VSR[VRT+32].dword[0] ← EXTZ64(vsrc.byte[index])
VSR[VRT+32].dword[1] ← 0x0000_0000_0000_0000
```

Let index be the contents of bits 59:63 of GPR[RC].

Let vsrc be the concatenation of the contents of VSR[VRA+32] and VSR[VRB+32].

The contents of byte element index of vsrc are zero-extended and placed into doubleword 0 of VSR[VRT+32].

The contents of doubleword 1 of VSR[VRT+32] are set to 0.

Special Registers Altered:

None

Vector Extract Double Unsigned Byte to VSR using GPR-specified Right-Index VA-form

vextdubvrx VRT,VRA,VRB,RC

4	VRT	VRA	VRB	RC	25
0	6	11	16	21	26 31

```
if MSR.VEC=0 then Vector_Unavailable()

index ← GPR[RC].bit[59:63]
vsrc.qword[0] ← VSR[VRA+32]
vsrc.qword[1] ← VSR[VRB+32]

VSR[VRT+32].dword[0] ← EXTZ64(vsrc.byte[31-index])
VSR[VRT+32].dword[1] ← 0x0000_0000_0000_0000
```

Let index be the contents of bits 59:63 of GPR[RC].

Let vsrc be the concatenation of the contents of VSR[VRA+32] and VSR[VRB+32].

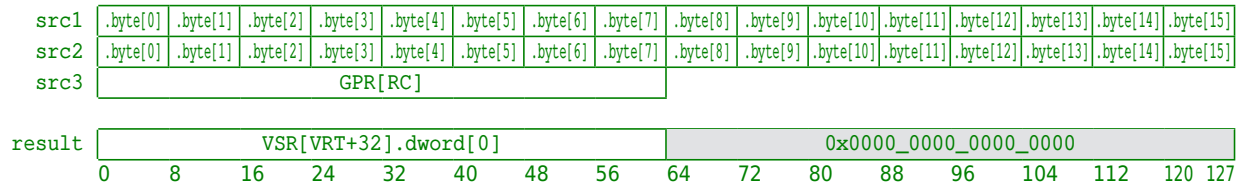
The contents of byte element 31-index of vsrc are zero-extended and placed into doubleword 0 of VSR[VRT+32].

The contents of doubleword 1 of VSR[VRT+32] are set to 0.

Special Registers Altered:

None

Register Data Layout for vextdubvlx & vextdubvrx



Vector Extract Double Unsigned Halfword to VSR using GPR-specified Left-Index VA-form

vextduhvlx VRT,VRA,VRB,RC

4	VRT	VRA	VRB	RC	26
0	6	11	16	21	26 31

```
if MSR.VEC=0 then Vector_Unavailable()

index ← GPR[RC].bit[59:63]
vsrc.qword[0] ← VSR[VRA+32]
vsrc.qword[1] ← VSR[VRB+32]

VSR[VRT+32].dword[0] ← EXTZ64(vsrc.byte[index:index+1])
VSR[VRT+32].dword[1] ← 0x0000_0000_0000_0000
```

Let `index` be the contents of bits 59:63 of `GPR[RC]`.

Let `vsrc` be the concatenation of the contents of `VSR[VRA+32]` and `VSR[VRB+32]`.

The contents of byte elements `index:index+1` of `vsrc` are zero-extended and placed into doubleword 0 of `VSR[VRT+32]`.

The contents of doubleword 1 of `VSR[VRT+32]` are set to 0.

If `index` is greater than 30, the result is undefined.

Special Registers Altered:

None

Vector Extract Double Unsigned Halfword to VSR using GPR-specified Right-Index VA-form

vextduhvrx VRT,VRA,VRB,RC

4	VRT	VRA	VRB	RC	27
0	6	11	16	21	26 31

```
if MSR.VEC=0 then Vector_Unavailable()

index ← GPR[RC].bit[59:63]
vsrc.qword[0] ← VSR[VRA+32]
vsrc.qword[1] ← VSR[VRB+32]

VSR[VRT+32].dword[0] ← EXTZ64(vsrc.byte[30-index:31-index])
VSR[VRT+32].dword[1] ← 0x0000_0000_0000_0000
```

Let `index` be the contents of bits 59:63 of `GPR[RC]`.

Let `vsrc` be the concatenation of the contents of `VSR[VRA+32]` and `VSR[VRB+32]`.

The contents of byte elements `30-index:31-index` of `vsrc` are zero-extended and placed into doubleword 0 of `VSR[VRT+32]`.

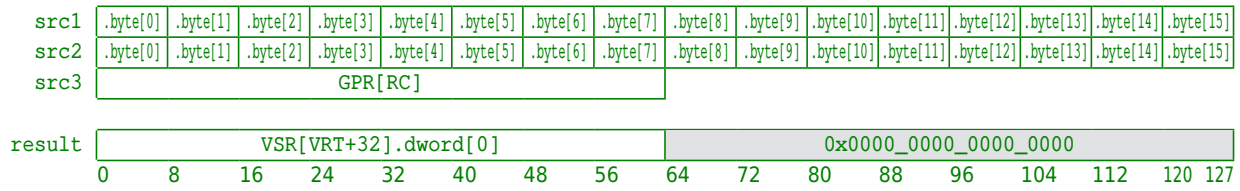
The contents of doubleword 1 of `VSR[VRT+32]` are set to 0.

If `index` is greater than 30, the result is undefined.

Special Registers Altered:

None

Register Data Layout for vextduhvlx & vextduhvrx



Vector Extract Double Unsigned Word to VSR using GPR-specified Left-Index VA-form

vextduwvlx VRT,VRA,VRB,RC

4	VRT	VRA	VRB	RC	28
0	6	11	16	21	26 31

```
if MSR.VEC=0 then Vector_Unavailable()

index ← GPR[RC].bit[59:63]
src.qword[0] ← VSR[VRA+32]
src.qword[1] ← VSR[VRB+32]

VSR[VRT+32].dword[0] ← EXTZ64(src.byte[index:index+3])
VSR[VRT+32].dword[1] ← 0x0000_0000_0000_0000
```

Let `index` be the contents of bits 59:63 of `GPR[RC]`.

Let `src` be the concatenation of the contents of `VSR[VRA+32]` and `VSR[VRB+32]`.

The contents of byte elements `index:index+3` of `src` are zero-extended and placed into doubleword 0 of `VSR[VRT+32]`.

The contents of doubleword 1 of `VSR[VRT+32]` are set to 0.

If `index` is greater than 28, the result is undefined.

Special Registers Altered:

None

Vector Extract Double Unsigned Word to VSR using GPR-specified Right-Index VA-form

vextduwvrx VRT,VRA,VRB,RC

4	VRT	VRA	VRB	RC	29
0	6	11	16	21	26 31

```
if MSR.VEC=0 then Vector_Unavailable()

index ← GPR[RC].bit[59:63]
src.qword[0] ← VSR[VRA+32]
src.qword[1] ← VSR[VRB+32]

VSR[VRT+32].dword[0] ←
   EXTZ64(src.byte[28-index:31-index])
VSR[VRT+32].dword[1] ← 0x0000_0000_0000_0000
```

Let `index` be the contents of bits 59:63 of `GPR[RC]`.

Let `src` be the concatenation of the contents of `VSR[VRA+32]` and `VSR[VRB+32]`.

The contents of byte elements `28-index:31-index` of `src` are zero-extended and placed into doubleword 0 of `VSR[VRT+32]`.

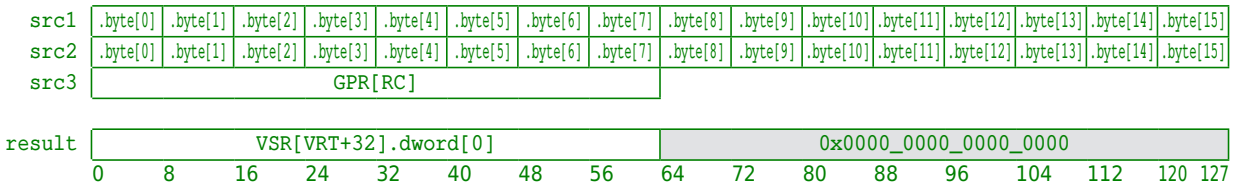
The contents of doubleword 1 of `VSR[VRT+32]` are set to 0.

If `index` is greater than 28, the result is undefined.

Special Registers Altered:

None

Register Data Layout for vextduwvlx & vextduwvrx



Vector Extract Double Doubleword to VSR using GPR-specified Left-Index VA-form

vextddvlx VRT,VRA,VRB,RC

4	VRT	VRA	VRB	RC	30
0	6	11	16	21	26 31

```
if MSR.VEC=0 then Vector_Unavailable()

index ← GPR[RC].bit[59:63]
src.qword[0] ← VSR[VRA+32]
src.qword[1] ← VSR[VRB+32]

VSR[VRT+32].dword[0] ← src.byte[index:index+7]
VSR[VRT+32].dword[1] ← 0x0000_0000_0000_0000
```

Let index be the contents of bits 59:63 of GPR[RC].

Let src be the concatenation of the contents of VSR[VRA+32] and VSR[VRB+32].

The contents of byte elements index:index+7 of src are placed into doubleword 0 of VSR[VRT+32].

The contents of doubleword 1 of VSR[VRT+32] are set to 0.

If index is greater than 24, the result is undefined.

Special Registers Altered:

None

Vector Extract Double Doubleword to VSR using GPR-specified Right-Index VA-form

vextddvrx VRT,VRA,VRB,RC

4	VRT	VRA	VRB	RC	31
0	6	11	16	21	26 31

```
if MSR.VEC=0 then Vector_Unavailable()

index ← GPR[RC].bit[59:63]
src.qword[0] ← VSR[VRA+32]
src.qword[1] ← VSR[VRB+32]

VSR[VRT+32].dword[0] ← src.byte[24-index:31-index]
VSR[VRT+32].dword[1] ← 0x0000_0000_0000_0000
```

Let index be the contents of bits 59:63 of GPR[RC].

Let src be the concatenation of the contents of VSR[VRA+32] and VSR[VRB+32].

The contents of byte elements 24-index:31-index of src are placed into doubleword 0 of VSR[VRT+32].

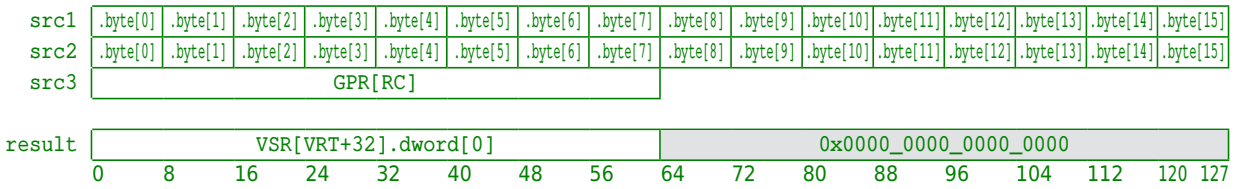
The contents of doubleword 1 of VSR[VRT+32] are set to 0.

If index is greater than 24, the result is undefined.

Special Registers Altered:

None

Register Data Layout for vextddvlx & vextddvrx



6.8.9 Vector Insert Element Instructions

6.8.9.1 Vector Insert Element from VSR Using Immediate-specified Index Instructions

Vector Insert Byte from VSR using immediate-specified index VX-form

vinsertb VRT,VRB,UIM

4	VRT	/	UIM	VRB	781
0	6	11	12	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

VSR[VRT+32].byte[UIM] ← VSR[VRB+32].byte[7]
```

The contents of byte element 7 of VSR[VRB+32] are placed into byte element UIM of VSR[VRT+32]. The contents of the remaining byte elements of VSR[VRT+32] are not modified.

Special Registers Altered:

None

Vector Insert Halfword from VSR using immediate-specified index VX-form

vinserth VRT,VRB,UIM

4	VRT	/	UIM	VRB	845
0	6	11	12	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

VSR[VRT+32].byte[UIM:UIM+1] ← VSR[VRB+32].hword[3]
```

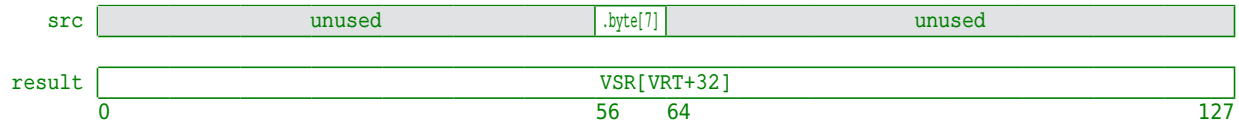
The contents of halfword element 3 of VSR[VRB+32] are placed into byte elements UIM:UIM+1 of VSR[VRT+32]. The contents of the remaining byte elements of VSR[VRT+32] are not modified.

If the value of UIM is greater than 14, the results are undefined.

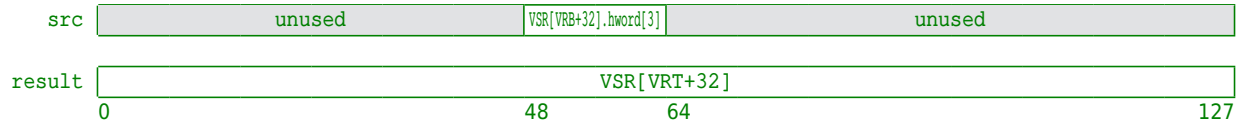
Special Registers Altered:

None

Register Data Layout for vinsertb



Register Data Layout for vinserth



Vector Insert Word from VSR using immediate-specified index VX-form

vinsertw VRT,VRB,UIM

4	VRT	/	UIM	VRB	909
0	6	11	12	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

VSR[VRT+32].byte[UIM:UIM+3] ← VSR[VRB+32].word[1]
```

The contents of word element 1 of VSR[VRB+32] are placed into byte elements UIM:UIM+3 of VSR[VRT+32]. The contents of the remaining byte elements of VSR[VRT+32] are not modified.

If the value of UIM is greater than 12, the results are undefined.

Special Registers Altered:

None

Vector Insert Doubleword from VSR using immediate-specified index VX-form

vinsertd VRT,VRB,UIM

4	VRT	/	UIM	VRB	973
0	6	11	12	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

VSR[VRT+32].byte[UIM:UIM+7] ← VSR[VRB+32].dword[0]
```

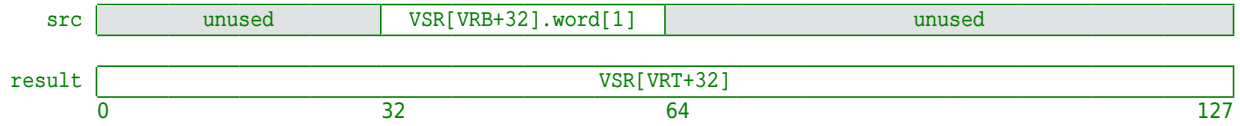
The contents of doubleword element 0 of VSR[VRB+32] are placed into byte elements UIM:UIM+7 of VSR[VRT+32]. The contents of the remaining byte elements of VSR[VRT+32] are not modified.

If the value of UIM is greater than 8, the results are undefined.

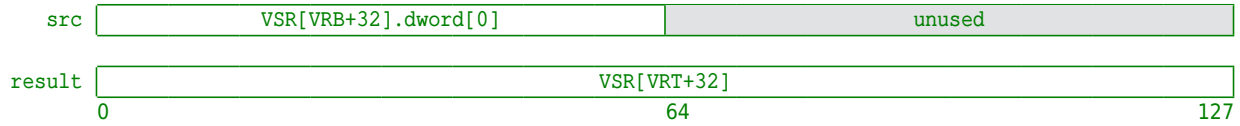
Special Registers Altered:

None

Register Data Layout for vinsertw



Register Data Layout for vinsertd



6.8.9.2 Vector Insert Element from GPR Using GPR-specified Index Instructions

Vector Insert Byte from GPR using GPR-specified Left-Index VX-form

vinsblx VRT,RA,RB

4	VRT	RA	RB	527
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

index ← GPR[RA].bit[60:63]
src.byte[0:15] ← 0

VSR[VRT+32].byte[index] ← GPR[RB].bit[56:63]
```

Let index be the contents of bits 60:63 of GPR[RA].

The contents of bits 56:63 of GPR[RB] are placed into byte element index of VSR[VRT+32].

All other byte elements of VSR[VRT+32] are not modified.

Special Registers Altered:

None

Vector Insert Byte from GPR using GPR-specified Right-Index VX-form

vinsbrx VRT,RA,RB

4	VRT	RA	RB	783
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

index ← GPR[RA].bit[60:63]
src.byte[0:15] ← 0

VSR[VRT+32].byte[15-index] ← GPR[RB].bit[56:63]
```

Let index be the contents of bits 60:63 of GPR[RA].

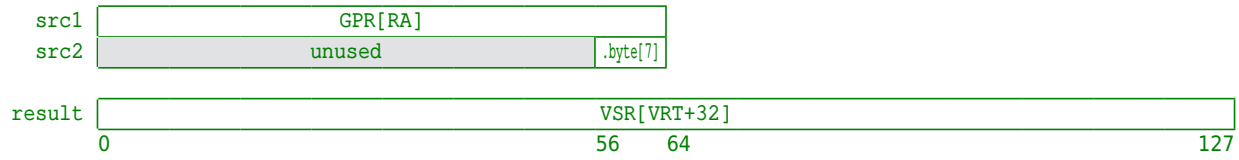
The contents of bits 56:63 of GPR[RB] are placed into byte element 15-index of VSR[VRT+32].

All other byte elements of VSR[VRT+32] are not modified.

Special Registers Altered:

None

Register Data Layout for vinsblx & vinsbrx



Vector Insert Halfword from GPR using GPR-specified Left-Index VX-form

vinshlx VRT,RA,RB

4	VRT	RA	RB	591
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

index ← GPR[RA].bit[60:63]
src.byte[0:15] ← 0

VSR[VRT+32].byte[index:index+1] ← GPR[RB].bit[48:63]
```

Let `index` be the contents of bits 60:63 of `GPR[RA]`.

The contents of bits 48:63 of `GPR[RB]` are placed into byte elements `index:index+1` of `VSR[VRT+32]`.

If `index` is greater than 14, the result is undefined.

Special Registers Altered:

None

Vector Insert Halfword from GPR using GPR-specified Right-Index VX-form

vinshrx VRT,RA,RB

4	VRT	RA	RB	847
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

index ← GPR[RA].bit[60:63]
src.byte[0:15] ← 0

VSR[VRT+32].byte[14-index:15-index] ←
   GPR[RB].bit[48:63]
```

Let `index` be the contents of bits 60:63 of `GPR[RA]`.

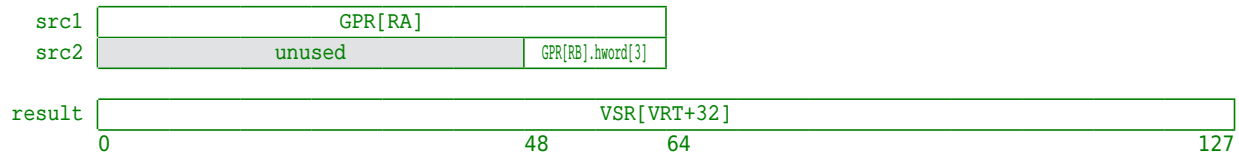
The contents of bits 48:63 of `GPR[RB]` are placed into byte elements `14-index:15-index` of `VSR[VRT+32]`.

If `index` is greater than 14, the result is undefined.

Special Registers Altered:

None

Register Data Layout for vinshlx & vinshrx



Vector Insert Word from GPR using GPR-specified Left-Index VX-form

vinswlx VRT,RA,RB

4	VRT	RA	RB	655
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

index ← GPR[RA].bit[60:63]

VSR[VRT+32].byte[index:index+3] ← GPR[RB].bit[32:63]
```

Let `index` be the contents of bits 60:63 of `GPR[RA]`.

The contents of bits 32:63 of `GPR[RB]` are placed into byte elements `index:index+3` of `VSR[VRT+32]`.

All other byte elements of `VSR[VRT+32]` are not modified.

If `index` is greater than 12, the result is undefined.

Special Registers Altered:

None

Vector Insert Word from GPR using GPR-specified Right-Index VX-form

vinswrx VRT,RA,RB

4	VRT	RA	RB	911
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

index ← GPR[RA].bit[60:63]

VSR[VRT+32].byte[12-index:15-index] ←
   GPR[RB].bit[32:63]
```

Let `index` be the contents of bits 60:63 of `GPR[RA]`.

The contents of bits 32:63 of `GPR[RB]` are placed into byte elements `12-index:15-index` of `VSR[VRT+32]`.

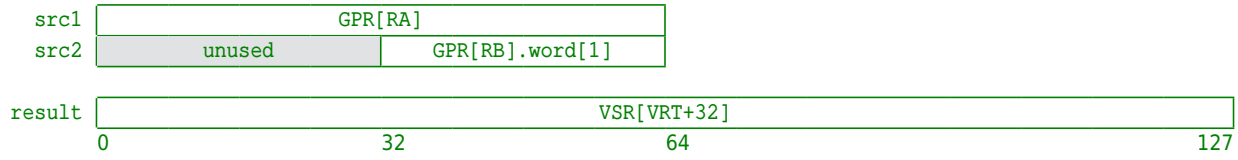
All other byte elements of `VSR[VRT+32]` are not modified.

If `index` is greater than 12, the result is undefined.

Special Registers Altered:

None

Register Data Layout for vinswlx & vinswrx



Vector Insert Doubleword from GPR using GPR-specified Left-Index VX-form

vinsdlx VRT,RA,RB

4	VRT	RA	RB	719
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

index ← GPR[RA].bit[60:63]

VSR[VRT+32].byte[index:index+7] ← GPR[RB]
```

Let index be the contents of bits 60:63 of GPR[RA].

The contents of GPR[RB] are placed into byte elements index:index+7 of VSR[VRT+32].

All other byte elements of VSR[VRT+32] are not modified.

If index is greater than 8, the result is undefined.

Special Registers Altered:

None

Vector Insert Doubleword from GPR using GPR-specified Right-Index VX-form

vinsdrx VRT,RA,RB

4	VRT	RA	RB	975
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

index ← GPR[RA].bit[60:63]

VSR[VRT+32].byte[8-index:15-index] ← GPR[RB]
```

Let index be the contents of bits 60:63 of GPR[RA].

The contents of GPR[RB] are placed into byte elements 8-index:15-index of VSR[VRT+32].

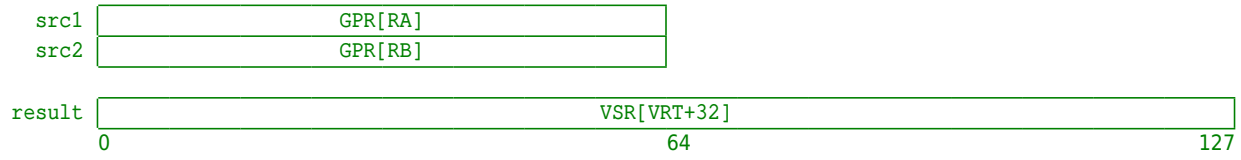
All other byte elements of VSR[VRT+32] are not modified.

If index is greater than 8, the result is undefined.

Special Registers Altered:

None

Register Data Layout for vinsdlx & vinsdrx



6.8.9.3 Vector Insert Element from GPR Using Immediate-specified Index Instructions

Vector Insert Word from GPR using immediate-specified index VX-form

vinsw VRT,RB,UIM

4	VRT	/	UIM	RB	207
0	6	11	12	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

VSR[VRT+32].byte[UIM:UIM+3] ← GPR[RB].bit[32:63]
```

The contents of bits 32:63 of `GPR[RB]` are placed into byte elements `UIM:UIM+3` of `VSR[VRT+32]`.

All other byte elements of `VSR[VRT+32]` are not modified.

If `UIM` is greater than 12, the result is undefined.

Special Registers Altered:

None

Vector Insert Doubleword from GPR using immediate-specified index VX-form

vinsd VRT,RB,UIM

4	VRT	/	UIM	RB	463
0	6	11	12	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

VSR[VRT+32].byte[UIM:UIM+7] ← GPR[RB]
```

The contents of `GPR[RB]` are placed into byte elements `UIM:UIM+7` of `VSR[VRT+32]`.

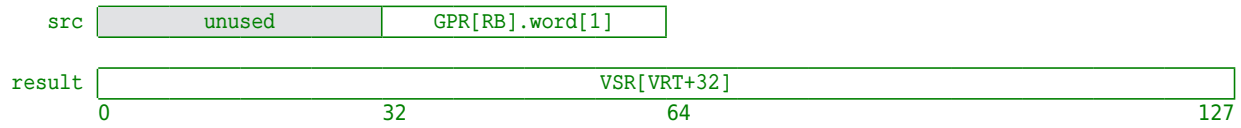
All other byte elements of `VSR[VRT+32]` are not modified.

If `UIM` is greater than 8, the result is undefined.

Special Registers Altered:

None

Register Data Layout for vinsw



Register Data Layout for vinsd

6.8.9.4 Vector Insert Element from VSR Using GPR-specified Index Instructions

Vector Insert Byte from VSR using GPR-specified Left-Index VX-form

vinsbvlx VRT,RA,VRB

4	VRT	RA	VRB	15
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

index ← GPR[RA].bit[60:63]

VSR[VRT+32].byte[index] ← VSR[VRB+32].bit[56:63]
```

Let index be the contents of bits 60:63 of GPR[RA].

The contents of bits 56:63 of VSR[VRB+32] are placed into byte element index of VSR[VRT+32].

All other byte elements of VSR[VRT+32] are not modified.

Special Registers Altered:

None

Vector Insert Byte from VSR using GPR-specified Right-Index VX-form

vinsbvrx VRT,RA,VRB

4	VRT	RA	VRB	271
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

index ← GPR[RA].bit[60:63]

VSR[VRT+32].byte[15-index] ← VSR[VRB+32].bit[56:63]
```

Let index be the contents of bits 60:63 of GPR[RA].

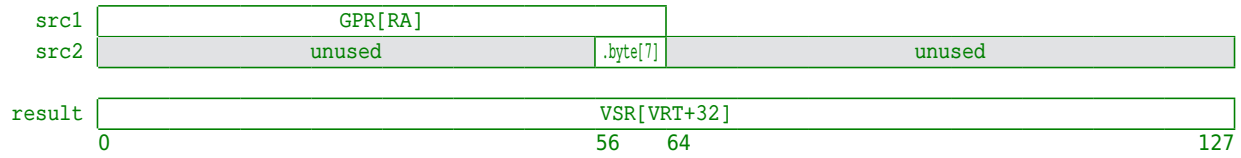
The contents of bits 56:63 of VSR[VRB+32] are placed into byte element 15-index of VSR[VRT+32].

All other byte elements of VSR[VRT+32] are not modified.

Special Registers Altered:

None

Register Data Layout for vinsbvlx & vinsbvrx



Vector Insert Halfword from VSR using GPR-specified Left-Index VX-form

vinshvlx VRT,RA,VRB

4	VRT	RA	VRB	79
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

index ← GPR[RA].bit[60:63]
src.byte[0:15] ← 0

VSR[VRT+32].byte[index:index+1] ←
   VSR[VRB+32].bit[48:63]
```

Let index be the contents of bits 60:63 of GPR[RA].

The contents of bits 48:63 of VSR[VRB+32] are placed into byte elements index:index+1 of VSR[VRT+32].

If index is greater than 14, the result is undefined.

Special Registers Altered:

None

Vector Insert Halfword from VSR using GPR-specified Right-Index VX-form

vinshvrx VRT,RA,VRB

4	VRT	RA	VRB	335
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

index ← GPR[RA].bit[60:63]
src.byte[0:15] ← 0

VSR[VRT+32].byte[14-index:15-index] ←
   VSR[VRB+32].bit[48:63]
```

Let index be the contents of bits 60:63 of GPR[RA].

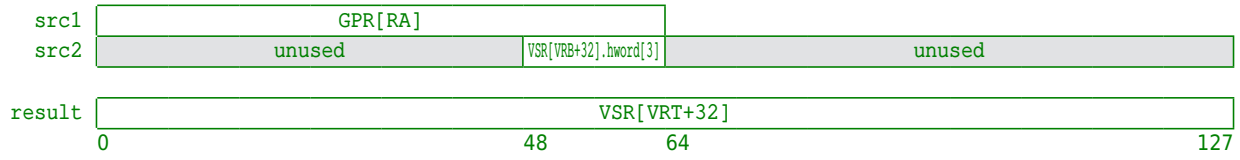
The contents of bits 48:63 of VSR[VRB+32] are placed into byte elements 14-index:15-index of VSR[VRT+32].

If index is greater than 14, the result is undefined.

Special Registers Altered:

None

Register Data Layout for vinshvlx & vinshvrx



Vector Insert Word from VSR using GPR-specified Left-Index VX-form

vinswvlx VRT,RA,VRB

4	VRT	RA	VRB	143
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

index ← GPR[RA].bit[60:63]

VSR[VRT+32].byte[index:index+3] ←
   VSR[VRB+32].bit[32:63]
```

Let index be the contents of bits 60:63 of GPR[RA].

The contents of bits 32:63 of VSR[VRB+32] are placed into byte elements index:index+3 of VSR[VRT+32].

All other byte elements of VSR[VRT+32] are not modified.

If index is greater than 12, the result is undefined.

Special Registers Altered:

None

Vector Insert Word from VSR using GPR-specified Right-Index VX-form

vinswvrx VRT,RA,VRB

4	VRT	RA	VRB	399
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

index ← GPR[RA].bit[60:63]

VSR[VRT+32].byte[12-index:15-index] ←
   VSR[VRB+32].bit[32:63]
```

Let index be the contents of bits 60:63 of GPR[RA].

The contents of bits 32:63 of VSR[VRB+32] are placed into byte elements 12-index:15-index of VSR[VRT+32].

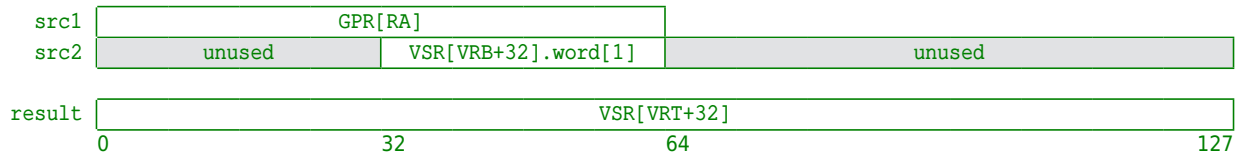
All other byte elements of VSR[VRT+32] are not modified.

If index is greater than 12, the result is undefined.

Special Registers Altered:

None

Register Data Layout for vinswvlx & vinswvrx



6.9 Vector Integer Instructions

6.9.1 Vector Integer Arithmetic Instructions

6.9.1.1 Vector Integer Add Instructions

Vector Add & write Carry-out Unsigned Word VX-form

vaddcuw VRT,VRA,VRB

4	VRT	VRA	VRB	384
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
  src1 ← EXTZ(VSR[VRA+32].word[i])
  src2 ← EXTZ(VSR[VRB+32].word[i])

  VSR[VRT+32].word[i] ← CHOP32((src1 + src2) >> 32)
end
```

For each integer value `i` from 0 to 3, do the following.

The unsigned integer value in word element `i` in `VSR[VRA+32]` is added to the unsigned integer value in word element `i` in `VSR[VRB+32]`. The carry out of the 32-bit sum is zero-extended to 32 bits and placed into word element `i` of `VSR[VRT+32]`.

Special Registers Altered:

None

Vector Add Signed Byte Saturate VX-form

vaddsbs VRT,VRA,VRB

4	VRT	VRA	VRB	768
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 15
   src1 ← EXTS(VSR[VRA+32].byte[i])
   src2 ← EXTS(VSR[VRB+32].byte[i])

   VSR[VRT+32].byte[i] ← si8_CLAMP(src1 + src2)
end
```

For each integer value `i` from 0 to 15, do the following.

The signed integer value in byte element `i` of `VSR[VRA+32]` is added to the signed integer value in byte element `i` of `VSR[VRB+32]`.

- If the sum is greater than $2^7 - 1$, the result saturates to $2^7 - 1$ and `SAT` is set to 1.
- If the sum is less than -2^7, the result saturates to -2^7 and `SAT` is set to 1.

The result is placed into byte element `i` of `VSR[VRT+32]`.

Special Registers Altered:

Register	*Field(s)*
VSCR	SAT

Register Data Layout for vaddcuw

src1	VSR[VRA+32].word[0]	VSR[VRA+32].word[1]	VSR[VRA+32].word[2]	VSR[VRA+32].word[3]
src2	VSR[VRB+32].word[0]	VSR[VRB+32].word[1]	VSR[VRB+32].word[2]	VSR[VRB+32].word[3]
result	VSR[VRT+32].word[0]	VSR[VRT+32].word[1]	VSR[VRT+32].word[2]	VSR[VRT+32].word[3]
	0	32	64	96 127

Register Data Layout for vaddsbs

src1	.byte[0]	.byte[1]	.byte[2]	.byte[3]	.byte[4]	.byte[5]	.byte[6]	.byte[7]	.byte[8]	.byte[9]	.byte[10]	.byte[11]	.byte[12]	.byte[13]	.byte[14]	.byte[15]
src2	.byte[0]	.byte[1]	.byte[2]	.byte[3]	.byte[4]	.byte[5]	.byte[6]	.byte[7]	.byte[8]	.byte[9]	.byte[10]	.byte[11]	.byte[12]	.byte[13]	.byte[14]	.byte[15]
result	.byte[0]	.byte[1]	.byte[2]	.byte[3]	.byte[4]	.byte[5]	.byte[6]	.byte[7]	.byte[8]	.byte[9]	.byte[10]	.byte[11]	.byte[12]	.byte[13]	.byte[14]	.byte[15]
	0	8	16	24	32	40	48	56	64	72	80	88	96	104	112	120 127

Vector Add Signed Halfword Saturate VX-form

vaddshs VRT,VRA,VRB

4	VRT	VRA	VRB	832
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 7
   src1 ← EXTS(VSR[VRA+32].hword[i])
   src2 ← EXTS(VSR[VRB+32].hword[i])

   VSR[VRT+32].hword[i] ← si16_CLAMP(src1 + src2)
end
```

For each integer value `i` from 0 to 7, do the following.

The signed integer value in halfword element `i` of `VSR[VRA+32]` is added to the signed integer value in halfword element `i` of `VSR[VRB+32]`.

- If the sum is greater than $2^{15}-1$, the result saturates to $2^{15}-1$ and `SAT` is set to 1.
- If the sum is less than -2^{15}, the result saturates to -2^{15} and `SAT` is set to 1.

The result is placed into halfword element `i` of `VSR[VRT+32]`.

Special Registers Altered:

Register	*Field(s)*
VSCR	SAT

Vector Add Signed Word Saturate VX-form

vaddsws VRT,VRA,VRB

4	VRT	VRA	VRB	896
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
   src1 ← EXTS(VSR[VRA].word[i])
   src2 ← EXTS(VSR[VRB].word[i])

   VSR[VRT+32].word[i] ← si32_CLAMP(src1 + src2)
end
```

For each integer value `i` from 0 to 3, do the following.

The signed integer value in word element `i` of `VSR[VRA+32]` is added to the signed integer value in word element `i` of `VSR[VRB+32]`.

- If the sum is greater than $2^{31}-1$, the result saturates to $2^{31}-1$ and `SAT` is set to 1.
- If the sum is less than -2^{31}, the result saturates to -2^{31} and `SAT` is set to 1.

The result is placed into word element `i` of `VSR[VRT+32]`.

Special Registers Altered:

Register	*Field(s)*
VSCR	SAT

Register Data Layout for vaddshs

src1	VSR[VRA+32].hword[0]	VSR[VRA+32].hword[1]	VSR[VRA+32].hword[2]	VSR[VRA+32].hword[3]	VSR[VRA+32].hword[4]	VSR[VRA+32].hword[5]	VSR[VRA+32].hword[6]	VSR[VRA+32].hword[7]
src2	VSR[VRB+32].hword[0]	VSR[VRB+32].hword[1]	VSR[VRB+32].hword[2]	VSR[VRB+32].hword[3]	VSR[VRB+32].hword[4]	VSR[VRB+32].hword[5]	VSR[VRB+32].hword[6]	VSR[VRB+32].hword[7]
result	VSR[VRT+32].hword[0]	VSR[VRT+32].hword[1]	VSR[VRT+32].hword[2]	VSR[VRT+32].hword[3]	VSR[VRT+32].hword[4]	VSR[VRT+32].hword[5]	VSR[VRT+32].hword[6]	VSR[VRT+32].hword[7]
	0	16	32	48	64	80	96	112 127

Register Data Layout for vaddsws

src1	VSR[VRA+32].word[0]	VSR[VRA+32].word[1]	VSR[VRA+32].word[2]	VSR[VRA+32].word[3]
src2	VSR[VRB+32].word[0]	VSR[VRB+32].word[1]	VSR[VRB+32].word[2]	VSR[VRB+32].word[3]
result	VSR[VRT+32].word[0]	VSR[VRT+32].word[1]	VSR[VRT+32].word[2]	VSR[VRT+32].word[3]
	0	32	64	96 127

Vector Add Unsigned Byte Modulo VX-form

vaddubm VRT,VRA,VRB

4	VRT	VRA	VRB	0
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 15
   src1 ← EXTZ(VSR[VRA+32].byte[i])
   src2 ← EXTZ(VSR[VRB+32].byte[i])

   VSR[VRT+32].byte[i] ← CHOP8(src1 + src2)
end
```

For each integer value `i` from 0 to 15, do the following.

The integer value in byte element `i` of `VSR[VRA+32]` is added to the integer value in byte element `i` of `VSR[VRB+32]`.

The low-order 8 bits of the result are placed into byte element `i` of `VSR[VRT+32]`.

Special Registers Altered:

None

Programming Note

vaddubm can be used for unsigned or signed integers.

Vector Add Unsigned Halfword Modulo VX-form

vadduhm VRT,VRA,VRB

4	VRT	VRA	VRB	64
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 7
   src1 ← EXTZ(VSR[VRA+32].hword[i])
   src2 ← EXTZ(VSR[VRB+32].hword[i])

   VSR[VRT+32].hword[i] ← CHOP16(src1 + src2)
end
```

For each integer value `i` from 0 to 7, do the following.

The integer value in halfword element `i` of `VSR[VRA+32]` is added to the integer value in halfword element `i` of `VSR[VRB+32]`.

The low-order 16 bits of the result are placed into halfword element i of `VSR[VRT+32]`.

Special Registers Altered:

None

Programming Note

vadduhm can be used for unsigned or signed integers.

Register Data Layout for vaddubm

src1	.byte[0]	.byte[1]	.byte[2]	.byte[3]	.byte[4]	.byte[5]	.byte[6]	.byte[7]	.byte[8]	.byte[9]	.byte[10]	.byte[11]	.byte[12]	.byte[13]	.byte[14]	.byte[15]
src2	.byte[0]	.byte[1]	.byte[2]	.byte[3]	.byte[4]	.byte[5]	.byte[6]	.byte[7]	.byte[8]	.byte[9]	.byte[10]	.byte[11]	.byte[12]	.byte[13]	.byte[14]	.byte[15]
result	.byte[0]	.byte[1]	.byte[2]	.byte[3]	.byte[4]	.byte[5]	.byte[6]	.byte[7]	.byte[8]	.byte[9]	.byte[10]	.byte[11]	.byte[12]	.byte[13]	.byte[14]	.byte[15]
	0	8	16	24	32	40	48	56	64	72	80	88	96	104	112	120 127

Register Data Layout for vadduhm

src1	VSR[VRA+32].hword[0]	VSR[VRA+32].hword[1]	VSR[VRA+32].hword[2]	VSR[VRA+32].hword[3]	VSR[VRA+32].hword[4]	VSR[VRA+32].hword[5]	VSR[VRA+32].hword[6]	VSR[VRA+32].hword[7]
src2	VSR[VRB+32].hword[0]	VSR[VRB+32].hword[1]	VSR[VRB+32].hword[2]	VSR[VRB+32].hword[3]	VSR[VRB+32].hword[4]	VSR[VRB+32].hword[5]	VSR[VRB+32].hword[6]	VSR[VRB+32].hword[7]
result	VSR[VRT+32].hword[0]	VSR[VRT+32].hword[1]	VSR[VRT+32].hword[2]	VSR[VRT+32].hword[3]	VSR[VRT+32].hword[4]	VSR[VRT+32].hword[5]	VSR[VRT+32].hword[6]	VSR[VRT+32].hword[7]
	0	16	32	48	64	80	96	112 127

Vector Add Unsigned Word Modulo VX-form

vadduwm VRT,VRA,VRB

4	VRT	VRA	VRB	128
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
   src1 ← EXTZ(VSR[VRA+32].word[i])
   src2 ← EXTZ(VSR[VRB+32].word[i])

   VSR[VRT+32].word[i] ← CHOP32(src1 + src2)
end
```

For each integer value `i` from 0 to 3, do the following.

The integer value in word element `i` of `VSR[VRA+32]` is added to the integer value in word element `i` of `VSR[VRB+32]`.

The low-order 32 bits of the result are placed into word element `i` of `VSR[VRT+32]`.

Special Registers Altered:

None

Programming Note

vadduwm can be used for unsigned or signed integers.

Vector Add Unsigned Doubleword Modulo VX-form

vaddudm VRT,VRA,VRB

4	VRT	VRA	VRB	192
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 1
   src1 ← EXTZ(VSR[VRA+32].dword[i])
   src2 ← EXTZ(VSR[VRB+32].dword[i])

   VSR[VRT+32].dword[i] ← CHOP64(src1 + src2)
end
```

For each integer value `i` from 0 to 1, do the following.

The integer value in doubleword element `i` of `VSR[VRB+32]` is added to the integer value in doubleword element `i` of `VSR[VRA+32]`.

The low-order 64 bits of the result are placed into doubleword element `i` of `VSR[VRT+32]`.

Special Registers Altered:

None

Programming Note

vaddudm can be used for signed or unsigned integers.

Register Data Layout for vadduwm

src1	VSR[VRA+32].word[0]	VSR[VRA+32].word[1]	VSR[VRA+32].word[2]	VSR[VRA+32].word[3]
src2	VSR[VRB+32].word[0]	VSR[VRB+32].word[1]	VSR[VRB+32].word[2]	VSR[VRB+32].word[3]
result	VSR[VRT+32].word[0]	VSR[VRT+32].word[1]	VSR[VRT+32].word[2]	VSR[VRT+32].word[3]
	0	32	64	96 127

Register Data Layout for vaddudm

src1	VSR[VRA+32].dword[0]	VSR[VRA+32].dword[1]
src2	VSR[VRB+32].dword[0]	VSR[VRB+32].dword[1]
result	VSR[VRT+32].dword[0]	VSR[VRT+32].dword[1]
	0	64 127

Vector Add Unsigned Byte Saturate VX-form

vaddubs VRT,VRA,VRB

4	VRT	VRA	VRB	512
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 15
   src1 ← EXTZ(VSR[VRA+32].byte[i])
   src2 ← EXTZ(VSR[VRB+32].byte[i])

   VSR[VRT+32].byte[i] ← ui8_CLAMP(src1 + src2)
end
```

For each integer value `i` from 0 to 15, do the following.

The unsigned integer value in byte element `i` of `VSR[VRA+32]` is added to the unsigned integer value in byte element `i` of `VSR[VRB+32]`.

- If the sum is greater than $2^8 - 1$, the result saturates to $2^8 - 1$ and `SAT` is set to 1.

The result is placed into byte element i of `VSR[VRT+32]`.

Special Registers Altered:

Register	*Field(s)*
VSCR	SAT

Vector Add Unsigned Halfword Saturate VX-form

vadduhs VRT,VRA,VRB

4	VRT	VRA	VRB	576
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 7
   src1 ← EXTZ(VSR[VRA+32].hword[i])
   src2 ← EXTZ(VSR[VRB+32].hword[i])

   VSR[VRT+32].hword[i] ← ui16_CLAMP(src1 + src2)
end
```

For each integer value `i` from 0 to 7, do the following.

The unsigned integer value in halfword element `i` of `VSR[VRA+32]` is added to the unsigned integer value in halfword element `i` of `VSR[VRB+32]`.

- If the sum is greater than $2^{16} - 1$, the result saturates to $2^{16} - 1$ and `SAT` is set to 1.

The result is placed into halfword element `i` of `VSR[VRT+32]`.

Special Registers Altered:

Register	*Field(s)*
VSCR	SAT

Register Data Layout for vaddubs

src1	.byte[0]	.byte[1]	.byte[2]	.byte[3]	.byte[4]	.byte[5]	.byte[6]	.byte[7]	.byte[8]	.byte[9]	.byte[10]	.byte[11]	.byte[12]	.byte[13]	.byte[14]	.byte[15]
src2	.byte[0]	.byte[1]	.byte[2]	.byte[3]	.byte[4]	.byte[5]	.byte[6]	.byte[7]	.byte[8]	.byte[9]	.byte[10]	.byte[11]	.byte[12]	.byte[13]	.byte[14]	.byte[15]
result	.byte[0]	.byte[1]	.byte[2]	.byte[3]	.byte[4]	.byte[5]	.byte[6]	.byte[7]	.byte[8]	.byte[9]	.byte[10]	.byte[11]	.byte[12]	.byte[13]	.byte[14]	.byte[15]
	0	8	16	24	32	40	48	56	64	72	80	88	96	104	112	120 127

Register Data Layout for vadduhs

src1	VSR[VRA+32].hword[0]	VSR[VRA+32].hword[1]	VSR[VRA+32].hword[2]	VSR[VRA+32].hword[3]	VSR[VRA+32].hword[4]	VSR[VRA+32].hword[5]	VSR[VRA+32].hword[6]	VSR[VRA+32].hword[7]
src2	VSR[VRB+32].hword[0]	VSR[VRB+32].hword[1]	VSR[VRB+32].hword[2]	VSR[VRB+32].hword[3]	VSR[VRB+32].hword[4]	VSR[VRB+32].hword[5]	VSR[VRB+32].hword[6]	VSR[VRB+32].hword[7]
result	VSR[VRT+32].hword[0]	VSR[VRT+32].hword[1]	VSR[VRT+32].hword[2]	VSR[VRT+32].hword[3]	VSR[VRT+32].hword[4]	VSR[VRT+32].hword[5]	VSR[VRT+32].hword[6]	VSR[VRT+32].hword[7]
	0	16	32	48	64	80	96	112 127

Vector Add Unsigned Word Saturate VX-form

vadduws VRT,VRA,VRB

4	VRT	VRA	VRB	640
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
   src1 ← EXTZ(VSR[VRA+32].word[i])
   src2 ← EXTZ(VSR[VRB+32].word[i])

   VSR[VRT+32].word[i] ← ui32_CLAMP(src1 + src2)
end
```

For each integer value i from 0 to 3, do the following.

The unsigned integer value in word element `i` of `VSR[VRA+32]` is added to the unsigned integer value in word element `i` of `VSR[VRB+32]`.

- If the sum is greater than $2^{32} - 1$, the result saturates to $2^{32} - 1$ and `SAT` is set to 1.

The result is placed into word element `i` of `VSR[VRT+32]`.

Special Registers Altered:

Register	*Field(s)*
VSCR	SAT

Register Data Layout for vadduws

src1	VSR[VRA+32].word[0]	VSR[VRA+32].word[1]	VSR[VRA+32].word[2]	VSR[VRA+32].word[3]
src2	VSR[VRB+32].word[0]	VSR[VRB+32].word[1]	VSR[VRB+32].word[2]	VSR[VRB+32].word[3]
result	VSR[VRT+32].word[0]	VSR[VRT+32].word[1]	VSR[VRT+32].word[2]	VSR[VRT+32].word[3]
	0	32	64	96 127

Vector Add Unsigned Quadword Modulo VX-form

vadduqm VRT,VRA,VRB

4	VRT	VRA	VRB	256
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

src1 ← EXTZ(VSR[VRA+32])
src2 ← EXTZ(VSR[VRB+32])

VSR[VRT+32] ← CHOP128(src1 + src2)
```

Let src1 be the integer value in VSR[VRA+32].
Let src2 be the integer value in VSR[VRB+32].

src1 and src2 can be signed or unsigned integers.

The rightmost 128 bits of the sum of src1 and src2 are placed into VSR[VRT+32].

Special Registers Altered:

None

Vector Add Extended Unsigned Quadword Modulo VA-form

vaddeuqm VRT,VRA,VRB,VRC

4	VRT	VRA	VRB	VRC	60
0	6	11	16	21	26 31

```
if MSR.VEC=0 then Vector_Unavailable()

src1 ← EXTZ(VSR[VRA+32])
src2 ← EXTZ(VSR[VRB+32])
cin  ← EXTZ(VSR[VRC+32].bit[127])

VSR[VRT+32] ← CHOP128(src1 + src2 + cin)
```

Let src1 be the integer value in VSR[VRA+32].
Let src2 be the integer value in VSR[VRB+32].
Let cin be the integer value in bit 127 of VSR[VRC+32].

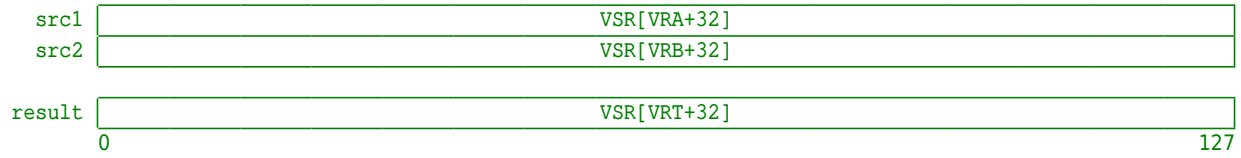
src1 and src2 can be signed or unsigned integers.

The rightmost 128 bits of the sum of src1, src2, and cin are placed into VSR[VRT+32].

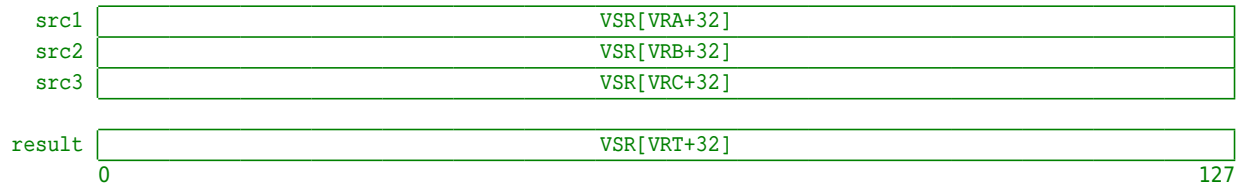
Special Registers Altered:

None

Register Data Layout for vadduqm



Register Data Layout for vaddeuqm



Vector Add & write Carry-out Unsigned Quadword VX-form

vaddcuq VRT,VRA,VRB

4	VRT	VRA	VRB	320
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

src1 ← EXTZ(VSR[VRA+32])
src2 ← EXTZ(VSR[VRB+32])
sum  ← EXTZ(src1) + EXTZ(src2)

VSR[VRT+32] ← EXTZ128((src1 + src2) >> 128)
```

Let src1 be the integer value in VSR[VRA+32].
Let src2 be the integer value in VSR[VRB+32].
src1 and src2 can be signed or unsigned integers.

The carry out of the sum of src1 and src2 is placed into VSR[VRT+32].

Special Registers Altered:

None

Vector Add Extended & write Carry-out Unsigned Quadword VA-form

vaddecuq VRT,VRA,VRB,VRC

4	VRT	VRA	VRB	VRC	61
0	6	11	16	21	26 31

```
if MSR.VEC=0 then Vector_Unavailable()

src1 ← EXTZ(VSR[VRA+32])
src2 ← EXTZ(VSR[VRB+32])
cin  ← EXTZ(VSR[VRC+32].bit[127])

VSR[VRT+32] ← EXTZ128((src1 + src2 + cin) >> 128)
```

Let src1 be the integer value in VSR[VRA+32].
Let src2 be the integer value in VSR[VRB+32].
Let cin be the integer value in bit 127 of VSR[VRC+32].

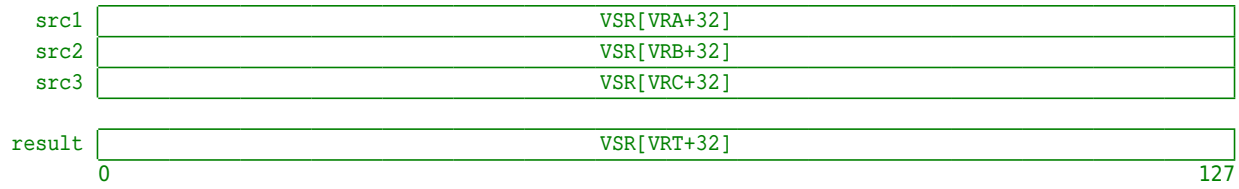
src1 and src2 can be signed or unsigned integers.

The carry out of the sum of src1, src2, and cin is placed into VSR[VRT+32].

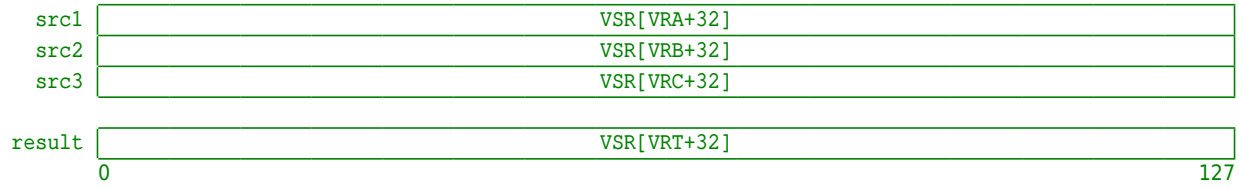
Special Registers Altered:

None

Register Data Layout for vaddcuq



Register Data Layout for vaddecuq



Programming Note

The *Vector Add Unsigned Quadword* instructions support efficient wide-integer addition. The following code sequence can be used to implement a 512-bit signed or unsigned add operation.

```
vadduqm     vS3,vA3,vB3         # bits 384:511 of sum
vaddcuq     vC3,vA3,vB3         # carry out of bit 384 of sum
vaddeuqm    vS2,vA2,vB2,vC3     # bits 256:383 of sum
vaddecuq    vC2,vA2,vB2,vC3     # carry out of bit 256 of sum
vaddeuqm    vS1,vA1,vB1,vC2     # bits 128:255 of sum
vaddecuq    vC1,vA1,vB1,vC2     # carry out of bit 128 of sum
vaddeuqm    vS0,vA0,vB0,vC1     # bits 0:127 of sum
```

6.9.1.2 Vector Integer Subtract Instructions

Vector Subtract & Write Carry-out Unsigned Word VX-form

vsubcuw VRT,VRA,VRB

4	VRT	VRA	VRB	1408
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
   src1 ← EXTZ(VSR[VRA+32].word[i])
   src2 ← EXTZ(¬VSR[VRB+32].word[i])

   VSR[VRT+32].word[i] ← EXTZ32((src1+src2+1) >> 32)
end
```

For each integer value i from 0 to 3, do the following.

The unsigned integer value in word element i of VSR[VRB+32] is subtracted from the unsigned integer value in word element i in VSR[VRA+32]. The complement of the borrow out of bit 0 of the 32-bit difference is zero-extended to 32 bits and placed into word element i of VSR[VRT+32].

Special Registers Altered:

None

Vector Subtract Signed Byte Saturate VX-form

vsubsbs VRT,VRA,VRB

4	VRT	VRA	VRB	1792
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 15
   src1 ← EXTS(VSR[VRA+32].byte[i])
   src2 ← EXTS(VSR[VRB+32].byte[i])

   VSR[VRT+32].byte[i] ← si8_CLAMP(src1 + ¬src2 + 1)
end
```

For each integer value i from 0 to 15, do the following.

The signed integer value in byte element i in VSR[VRB+32] is subtracted from the signed integer value in byte element i in VSR[VRA+32].

- If the intermediate result is greater than 127, the result saturates to 127 and SAT is set to 1.
- If the intermediate result is less than -128, the result saturates to -128 and SAT is set to 1.

The result is placed into byte element i of VSR[VRT+32].

Special Registers Altered:

Register	*Field(s)*
VSCR	SAT

Register Data Layout for vsubcuw

src1	VSR[VRA+32].word[0]	VSR[VRA+32].word[1]	VSR[VRA+32].word[2]	VSR[VRA+32].word[3]
src2	VSR[VRB+32].word[0]	VSR[VRB+32].word[1]	VSR[VRB+32].word[2]	VSR[VRB+32].word[3]
result	VSR[VRT+32].word[0]	VSR[VRT+32].word[1]	VSR[VRT+32].word[2]	VSR[VRT+32].word[3]
	0	32	64	96 127

Register Data Layout for vsubsbs

src1	.byte[0]	.byte[1]	.byte[2]	.byte[3]	.byte[4]	.byte[5]	.byte[6]	.byte[7]	.byte[8]	.byte[9]	.byte[10]	.byte[11]	.byte[12]	.byte[13]	.byte[14]	.byte[15]
src2	.byte[0]	.byte[1]	.byte[2]	.byte[3]	.byte[4]	.byte[5]	.byte[6]	.byte[7]	.byte[8]	.byte[9]	.byte[10]	.byte[11]	.byte[12]	.byte[13]	.byte[14]	.byte[15]
result	.byte[0]	.byte[1]	.byte[2]	.byte[3]	.byte[4]	.byte[5]	.byte[6]	.byte[7]	.byte[8]	.byte[9]	.byte[10]	.byte[11]	.byte[12]	.byte[13]	.byte[14]	.byte[15]
	0	8	16	24	32	40	48	56	64	72	80	88	96	104	112	120 127

Vector Subtract Signed Halfword Saturate VX-form

vsubshs VRT,VRA,VRB

4	VRT	VRA	VRB	1856
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 7
   src1 ← EXTS(VSR[VRA+32].hword[i])
   src2 ← EXTS(VSR[VRB+32].hword[i])

   VSR[VRT+32].hword[i] ← si16_CLAMP(src1 + ¬src2 + 1)
end
```

For each integer value `i` from 0 to 7, do the following.

The signed integer value in halfword element `i` in `VSR[VRB+32]` is subtracted from the signed integer value in halfword element `i` in `VSR[VRA+32]`.

- If the intermediate result is greater than $2^{15}-1$, the result saturates to $2^{15}-1$ and `SAT` is set to 1.
- If the intermediate result is less than -2^{15}, the result saturates to -2^{15} and `SAT` is set to 1.

The result is placed into halfword element i of `VSR[VRT+32]`.

Special Registers Altered:

Register	*Field(s)*
VSCR	SAT

Vector Subtract Signed Word Saturate VX-form

vsubsws VRT,VRA,VRB

4	VRT	VRA	VRB	1920
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
   src1 ← EXTS(VSR[VRA+32].word[i])
   src2 ← EXTS(VSR[VRB+32].word[i])

   VSR[VRT+32].word[i] ← si32_CLAMP(src1 + ¬src2 + 1)
end
```

For each integer value `i` from 0 to 3, do the following.

The signed integer value in word element `i` in `VSR[VRB+32]` is subtracted from the signed integer value in word element `i` in `VSR[VRA+32]`.

- If the intermediate result is greater than $2^{31}-1$, the result saturates to $2^{31}-1$ and `SAT` is set to 1.
- If the intermediate result is less than -2^{31}, the result saturates to -2^{31} and `SAT` is set to 1.

The result is placed into word element i of `VSR[VRT+32]`.

Special Registers Altered:

Register	*Field(s)*
VSCR	SAT

Register Data Layout for vsubshs

src1	VSR[VRA+32].hword[0]	VSR[VRA+32].hword[1]	VSR[VRA+32].hword[2]	VSR[VRA+32].hword[3]	VSR[VRA+32].hword[4]	VSR[VRA+32].hword[5]	VSR[VRA+32].hword[6]	VSR[VRA+32].hword[7]
src2	VSR[VRB+32].hword[0]	VSR[VRB+32].hword[1]	VSR[VRB+32].hword[2]	VSR[VRB+32].hword[3]	VSR[VRB+32].hword[4]	VSR[VRB+32].hword[5]	VSR[VRB+32].hword[6]	VSR[VRB+32].hword[7]
result	VSR[VRT+32].hword[0]	VSR[VRT+32].hword[1]	VSR[VRT+32].hword[2]	VSR[VRT+32].hword[3]	VSR[VRT+32].hword[4]	VSR[VRT+32].hword[5]	VSR[VRT+32].hword[6]	VSR[VRT+32].hword[7]
	0	16	32	48	64	80	96	112 127

Register Data Layout for vsubsws

src1	VSR[VRA+32].word[0]	VSR[VRA+32].word[1]	VSR[VRA+32].word[2]	VSR[VRA+32].word[3]
src2	VSR[VRB+32].word[0]	VSR[VRB+32].word[1]	VSR[VRB+32].word[2]	VSR[VRB+32].word[3]
result	VSR[VRT+32].word[0]	VSR[VRT+32].word[1]	VSR[VRT+32].word[2]	VSR[VRT+32].word[3]
	0	32	64	96 127

Vector Subtract Unsigned Byte Modulo VX-form

vsububm VRT,VRA,VRB

4	VRT	VRA	VRB	1024
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 15
   src1 ← EXTZ(VSR[VRA+32].byte[i])
   src2 ← EXTZ(VSR[VRB+32].byte[i])
   VSR[VRT+32].byte[i] ← CHOP8(src1 + ¬src2 + 1)
end
```

For each integer value `i` from 0 to 15, do the following.

The unsigned integer value in byte element `i` in `VSR[VRB+32]` is subtracted from the unsigned integer value in byte element `i` in `VSR[VRA+32]`.

The low-order 8 bits of the result are placed into byte element `i` of `VSR[VRT+32]`.

Special Registers Altered:

None

Vector Subtract Unsigned Halfword Modulo VX-form

vsubuhm VRT,VRA,VRB

4	VRT	VRA	VRB	1088
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 7
   src1 ← EXTZ(VSR[VRA+32].hword[i])
   src2 ← EXTZ(VSR[VRB+32].hword[i])
   VSR[VRT+32].hword[i] ← CHOP16(src1 + ¬src2 + 1)
end
```

For each integer value `i` from 0 to 7, do the following.

The unsigned integer value in halfword element `i` in `VSR[VRB+32]` is subtracted from the unsigned integer value in halfword element `i` in `VSR[VRA+32]`.

The low-order 16 bits of the result are placed into halfword element `i` of `VSR[VRT+32]`.

Special Registers Altered:

None

Register Data Layout for vsububm

src1	.byte[0]	.byte[1]	.byte[2]	.byte[3]	.byte[4]	.byte[5]	.byte[6]	.byte[7]	.byte[8]	.byte[9]	.byte[10]	.byte[11]	.byte[12]	.byte[13]	.byte[14]	.byte[15]
src2	.byte[0]	.byte[1]	.byte[2]	.byte[3]	.byte[4]	.byte[5]	.byte[6]	.byte[7]	.byte[8]	.byte[9]	.byte[10]	.byte[11]	.byte[12]	.byte[13]	.byte[14]	.byte[15]
result	.byte[0]	.byte[1]	.byte[2]	.byte[3]	.byte[4]	.byte[5]	.byte[6]	.byte[7]	.byte[8]	.byte[9]	.byte[10]	.byte[11]	.byte[12]	.byte[13]	.byte[14]	.byte[15]
	0	8	16	24	32	40	48	56	64	72	80	88	96	104	112	120 127

Register Data Layout for vsubuhm

src1	VSR[VRA+32].hword[0]	VSR[VRA+32].hword[1]	VSR[VRA+32].hword[2]	VSR[VRA+32].hword[3]	VSR[VRA+32].hword[4]	VSR[VRA+32].hword[5]	VSR[VRA+32].hword[6]	VSR[VRA+32].hword[7]
src2	VSR[VRB+32].hword[0]	VSR[VRB+32].hword[1]	VSR[VRB+32].hword[2]	VSR[VRB+32].hword[3]	VSR[VRB+32].hword[4]	VSR[VRB+32].hword[5]	VSR[VRB+32].hword[6]	VSR[VRB+32].hword[7]
result	VSR[VRT+32].hword[0]	VSR[VRT+32].hword[1]	VSR[VRT+32].hword[2]	VSR[VRT+32].hword[3]	VSR[VRT+32].hword[4]	VSR[VRT+32].hword[5]	VSR[VRT+32].hword[6]	VSR[VRT+32].hword[7]
	0	16	32	48	64	80	96	112 127

Vector Subtract Unsigned Word Modulo VX-form

vsubuwm VRT,VRA,VRB

4	VRT	VRA	VRB	1152
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
   src1 ← EXTZ(VSR[VRA+32].word[i])
   src2 ← EXTZ(VSR[VRB+32].word[i])
   VSR[VRT+32].word[i] ← CHOP32(src1 + ¬src2 + 1)
end
```

For each integer value `i` from 0 to 3, do the following.

The unsigned integer value in word element `i` in `VSR[VRB+32]` is subtracted from the unsigned integer value in word element `i` in `VSR[VRA+32]`.

The low-order 32 bits of the result are placed into word element `i` of `VSR[VRT+32]`.

Special Registers Altered:

None

Vector Subtract Unsigned Doubleword Modulo VX-form

vsubudm VRT,VRA,VRB

4	VRT	VRA	VRB	1216
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 1
   src1 ← EXTZ(VSR[VRA+32].dword[i])
   src2 ← EXTZ(VSR[VRB+32].dword[i])
   VSR[VRT+32].dword[i] ← CHOP64(src1 + ¬src2 + 1)
end
```

For each integer value `i` from 0 to 1, do the following.

The integer value in doubleword element `i` in `VSR[VRB+32]` is subtracted from the integer value in doubleword element `i` in `VSR[VRA+32]`.

The low-order 64 bits of the result are placed into doubleword element `i` of `VSR[VRT+32]`.

Special Registers Altered:

None

Programming Note

vsubudm can be used for signed or unsigned integers.

Register Data Layout for vsubuwm

src1	VSR[VRA+32].word[0]	VSR[VRA+32].word[1]	VSR[VRA+32].word[2]	VSR[VRA+32].word[3]
src2	VSR[VRB+32].word[0]	VSR[VRB+32].word[1]	VSR[VRB+32].word[2]	VSR[VRB+32].word[3]
result	VSR[VRT+32].word[0]	VSR[VRT+32].word[1]	VSR[VRT+32].word[2]	VSR[VRT+32].word[3]
	0	32	64	96 127

Register Data Layout for vsubudm

src1	VSR[VRA+32].dword[0]	VSR[VRA+32].dword[1]
src2	VSR[VRB+32].dword[0]	VSR[VRB+32].dword[1]
result	VSR[VRT+32].dword[0]	VSR[VRT+32].dword[1]
	0	64 127

Vector Subtract Unsigned Byte Saturate VX-form

vsububs VRT,VRA,VRB

4	VRT	VRA	VRB	1536
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 15
   src1 ← EXTZ(VSR[VRA+32].byte[i])
   src2 ← EXTZ(VSR[VRB+32].byte[i])
   VSR[VRT+32].byte[i] ← ui8_CLAMP(src1 + ¬src2 + 1)
end
```

For each integer value `i` from 0 to 15, do the following.

The unsigned integer value in byte element `i` of `VSR[VRB+32]` is subtracted from the unsigned integer value in byte element `i` of `VSR[VRA+32]`.

- If the intermediate result is less than 0, the result saturates to 0 and `SAT` is set to 1.

The result is placed into byte element `i` of `VSR[VRT+32]`.

Special Registers Altered:

Register	*Field(s)*
VSCR	SAT

Vector Subtract Unsigned Halfword Saturate VX-form

vsubuhs VRT,VRA,VRB

4	VRT	VRA	VRB	1600
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 7
   src1 ← EXTZ(VSR[VRA+32].hword[i])
   src2 ← EXTZ(VSR[VRB+32].hword[i])
   VSR[VRT+32].hword[i] ← ui16_CLAMP(src1 + ¬src2 + 1)
   VSCR.SAT ← VSCR.SAT | sat_flag
end
```

For each integer value `i` from 0 to 7, do the following.

The unsigned integer value in halfword element `i` of `VSR[VRB+32]` is subtracted from the unsigned integer value in halfword element `i` of `VSR[VRA+32]`.

- If the intermediate result is less than 0, the result saturates to 0 and `SAT` is set to 1.

The result is placed into halfword element `i` of `VSR[VRT+32]`.

Special Registers Altered:

Register	*Field(s)*
VSCR	SAT

Register Data Layout for vsububs

src1	.byte[0]	.byte[1]	.byte[2]	.byte[3]	.byte[4]	.byte[5]	.byte[6]	.byte[7]	.byte[8]	.byte[9]	.byte[10]	.byte[11]	.byte[12]	.byte[13]	.byte[14]	.byte[15]
src2	.byte[0]	.byte[1]	.byte[2]	.byte[3]	.byte[4]	.byte[5]	.byte[6]	.byte[7]	.byte[8]	.byte[9]	.byte[10]	.byte[11]	.byte[12]	.byte[13]	.byte[14]	.byte[15]
result	.byte[0]	.byte[1]	.byte[2]	.byte[3]	.byte[4]	.byte[5]	.byte[6]	.byte[7]	.byte[8]	.byte[9]	.byte[10]	.byte[11]	.byte[12]	.byte[13]	.byte[14]	.byte[15]
	0	8	16	24	32	40	48	56	64	72	80	88	96	104	112	120 127

Register Data Layout for vsubuhs

src1	VSR[VRA+32].hword[0]	VSR[VRA+32].hword[1]	VSR[VRA+32].hword[2]	VSR[VRA+32].hword[3]	VSR[VRA+32].hword[4]	VSR[VRA+32].hword[5]	VSR[VRA+32].hword[6]	VSR[VRA+32].hword[7]
src2	VSR[VRB+32].hword[0]	VSR[VRB+32].hword[1]	VSR[VRB+32].hword[2]	VSR[VRB+32].hword[3]	VSR[VRB+32].hword[4]	VSR[VRB+32].hword[5]	VSR[VRB+32].hword[6]	VSR[VRB+32].hword[7]
result	VSR[VRT+32].hword[0]	VSR[VRT+32].hword[1]	VSR[VRT+32].hword[2]	VSR[VRT+32].hword[3]	VSR[VRT+32].hword[4]	VSR[VRT+32].hword[5]	VSR[VRT+32].hword[6]	VSR[VRT+32].hword[7]
	0	16	32	48	64	80	96	112 127

Vector Subtract Unsigned Word Saturate VX-form

vsubuws VRT,VRA,VRB

4	VRT	VRA	VRB	1664
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
   src1 ← EXTZ(VSR[VRA+32].word[i])
   src2 ← EXTZ(VSR[VRB+32].word[i])
   VSR[VRT+32].word[i] ← ui32_CLAMP(src1 + ¬src2 + 1)
end
```

For each integer value `i` from 0 to 3, do the following.

The unsigned integer value in word element `i` of `VSR[VRB+32]` is subtracted from the unsigned integer value in word element `i` of `VSR[VRA+32]`.

- If the intermediate result is less than 0, the result saturates to 0 and `SAT` is set to 1.

The result is placed into word element `i` of `VSR[VRT+32]`.

Special Registers Altered:

Register	*Field(s)*
VSCR	SAT

Register Data Layout for vsubuws

src1	VSR[VRA+32].word[0]	VSR[VRA+32].word[1]	VSR[VRA+32].word[2]	VSR[VRA+32].word[3]
src2	VSR[VRB+32].word[0]	VSR[VRB+32].word[1]	VSR[VRB+32].word[2]	VSR[VRB+32].word[3]
result	VSR[VRT+32].word[0]	VSR[VRT+32].word[1]	VSR[VRT+32].word[2]	VSR[VRT+32].word[3]
	0	32	64	96 127

Vector Subtract Unsigned Quadword Modulo VX-form

vsubuqm VRT,VRA,VRB

4	VRT	VRA	VRB	1280
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

src1 ← EXTZ(VSR[VRA+32])
src2 ← EXTZ(¬VSR[VRB+32])

VSR[VRT+32] ← CHOP128(src1 + src2 + 1)
```

Let src1 be the integer value in VSR[VRA+32].
Let src2 be the integer value in VSR[VRB+32].

src1 and src2 can be signed or unsigned integers.

The rightmost 128 bits of the sum of src1, the one's complement of src2, and the value 1 are placed into VSR[VRT+32].

Special Registers Altered:

None

Vector Subtract Extended Unsigned Quadword Modulo VA-form

vsubeuqm VRT,VRA,VRB,VRC

4	VRT	VRA	VRB	VRC	62
0	6	11	16	21	26 31

```
if MSR.VEC=0 then Vector_Unavailable()

src1 ← EXTZ(VSR[VRA+32])
src2 ← EXTZ(¬VSR[VRB+32])
cin  ← EXTZ(VSR[VRC+32].bit[127])

VSR[VRT+32] ← CHOP128(src1 + src2 + cin)
```

Let src1 be the integer value in VSR[VRA+32].
Let src2 be the integer value in VSR[VRB+32].
Let cin be the integer value in bit 127 of VSR[VRC+32].

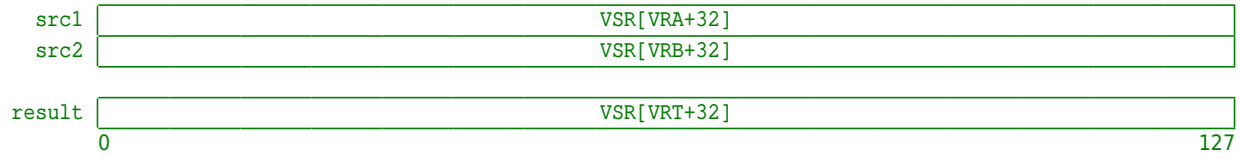
src1 and src2 can be signed or unsigned integers.

The rightmost 128 bits of the sum of src1, the one's complement of src2, and cin are placed into VSR[VRT+32].

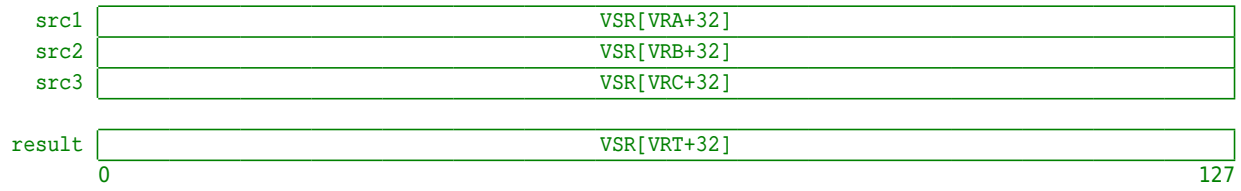
Special Registers Altered:

None

Register Data Layout for vsubuqm



Register Data Layout for vsubeuqm



Vector Subtract & write Carry-out Unsigned Quadword VX-form

vsubcuq VRT,VRA,VRB

4	VRT	VRA	VRB	1344
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

src1 ← EXTZ(VSR[VRA+32])
src2 ← EXTZ(¬VSR[VRB+32])

VSR[VRT+32] ← CHOP128((src1 + src2 + 1) >> 128)
```

Let src1 be the integer value in VSR[VRA+32].
Let src2 be the integer value in VSR[VRB+32].

src1 and src2 can be signed or unsigned integers.

The carry out of the sum of src1, the one's complement of src2, and the value 1 is placed into VSR[VRT+32].

Special Registers Altered:

None

Vector Subtract Extended & write Carry-out Unsigned Quadword VA-form

vsubecuq VRT,VRA,VRB,VRC

4	VRT	VRA	VRB	VRC	63
0	6	11	16	21	26 31

```
if MSR.VEC=0 then Vector_Unavailable()

src1 ← EXTZ(VSR[VRA+32])
src2 ← EXTZ(¬VSR[VRB+32])
cin  ← EXTZ(VSR[VRC+32].bit[127])

VSR[VRT+32] ← CHOP128((src1 + src2 + cin) >> 128)
```

Let src1 be the integer value in VSR[VRA+32].
Let src2 be the integer value in VSR[VRB+32].
Let cin be the integer value in bit 127 of VSR[VRC+32].

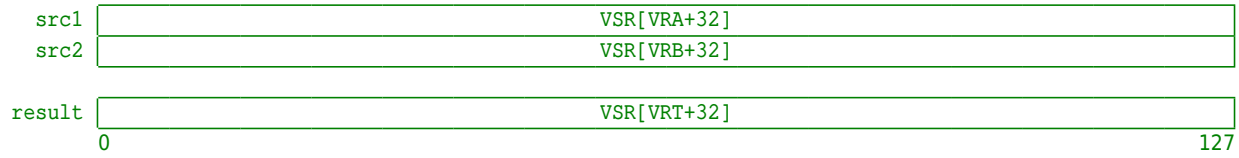
src1 and src2 can be signed or unsigned integers.

The carry out of the sum of src1, the one's complement of src2, and cin are placed into VSR[VRT+32].

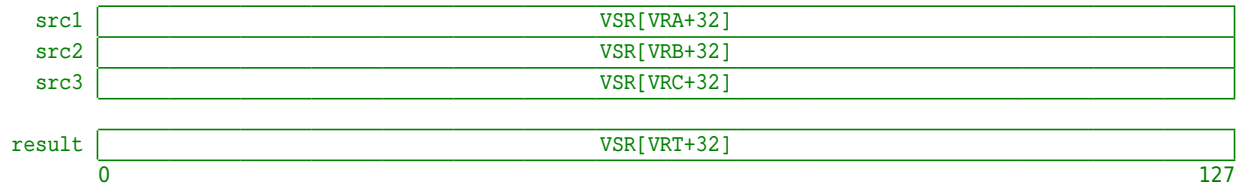
Special Registers Altered:

None

Register Data Layout for vsubcuq



Register Data Layout for vsubecuq



Programming Note

The *Vector Subtract Unsigned Quadword* instructions support efficient wide-integer subtraction. The following code sequence can be used to implement a 512-bit signed or unsigned subtract operation.

```
vsubuqm     vS3,vA3,vB3         # bits 384:511 of difference
vsubcuq     vC3,vA3,vB3         # carry out of bit 384 of difference
vsubeuqm    vS2,vA2,vB2,vC3     # bits 256:383 of difference
vsubecuq    vC2,vA2,vB2,vC3     # carry out of bit 256 of difference
vsubeuqm    vS1,vA1,vB1,vC2     # bits 128:255 of difference
vsubecuq    vC1,vA1,vB1,vC2     # carry out of bit 128 of difference
vsubeuqm    vS0,vA0,vB0,vC1     # bits 0:127 of difference
```

6.9.1.3 Vector Integer Multiply Instructions

Vector Multiply Even Signed Byte VX-form

vmulesb VRT,VRA,VRB

4	VRT	VRA	VRB	776
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 7
   src1 ← EXTS(VSR[VRA+32].byte[2×i])
   src2 ← EXTS(VSR[VRB+32].byte[2×i])

   VSR[VRT+32].hword[i] ← CHOP16(src1 × src2)
end
```

For each integer value `i` from 0 to 7, do the following.

The signed integer value in byte element `i×2` of `VSR[VRA+32]` is multiplied by the signed integer value in byte element `i×2` of `VSR[VRB+32]`.

The 16-bit product is placed into halfword element `i` of `VSR[VRT+32]`.

Special Registers Altered:

None

Vector Multiply Odd Signed Byte VX-form

vmulosb VRT,VRA,VRB

4	VRT	VRA	VRB	264
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 7
   src1 ← EXTS(VSR[VRA+32].byte[2×i+1])
   src2 ← EXTS(VSR[VRB+32].byte[2×i+1])

   VSR[VRT+32].hword[i] ← CHOP16(src1 × src2)
end
```

For each integer value `i` from 0 to 7, do the following.

The signed integer value in byte element `i×2+1` of `VSR[VRA+32]` is multiplied by the signed integer value in byte element `i×2+1` of `VSR[VRB+32]`.

The 16-bit product is placed into halfword element `i` of `VSR[VRT+32]`.

Special Registers Altered:

None

Register Data Layout for vmulesb

src1	.byte[0]	unused	.byte[2]	unused	.byte[4]	unused	.byte[6]	unused	.byte[8]	unused	.byte[10]	unused	.byte[12]	unused	.byte[14]	unused
src2	.byte[0]	unused	.byte[2]	unused	.byte[4]	unused	.byte[6]	unused	.byte[8]	unused	.byte[10]	unused	.byte[12]	unused	.byte[14]	unused

result	VSR[VRT+32].hword[0]		VSR[VRT+32].hword[1]		VSR[VRT+32].hword[2]		VSR[VRT+32].hword[3]		VSR[VRT+32].hword[4]		VSR[VRT+32].hword[5]		VSR[VRT+32].hword[6]		VSR[VRT+32].hword[7]	
	0	8	16	24	32	40	48	56	64	72	80	88	96	104	112	120 127

Register Data Layout for vmulosb

src1	unused	.byte[1]	unused	.byte[3]	unused	.byte[5]	unused	.byte[7]	unused	.byte[9]	unused	.byte[11]	unused	.byte[13]	unused	.byte[15]
src2	unused	.byte[1]	unused	.byte[3]	unused	.byte[5]	unused	.byte[7]	unused	.byte[9]	unused	.byte[11]	unused	.byte[13]	unused	.byte[15]

result	VSR[VRT+32].hword[0]		VSR[VRT+32].hword[1]		VSR[VRT+32].hword[2]		VSR[VRT+32].hword[3]		VSR[VRT+32].hword[4]		VSR[VRT+32].hword[5]		VSR[VRT+32].hword[6]		VSR[VRT+32].hword[7]	
	0	8	16	24	32	40	48	56	64	72	80	88	96	104	112	120 127

Vector Multiply Even Unsigned Byte VX-form

vmuleub VRT,VRA,VRB

4	VRT	VRA	VRB	520
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 7
   src1 ← EXTZ(VSR[VRA+32].byte[2×i])
   src2 ← EXTZ(VSR[VRB+32].byte[2×i])

   VSR[VRT+32].hword[i] ← CHOP16(src1 × src2)
end
```

For each integer value `i` from 0 to 7, do the following.

The unsigned integer value in byte element `i×2` of `VSR[VRA+32]` is multiplied by the unsigned integer value in byte element `i×2` of `VSR[VRB+32]`.

The 16-bit product is placed into halfword element `i` of `VSR[VRT+32]`.

Special Registers Altered:

None

Vector Multiply Odd Unsigned Byte VX-form

vmuloub VRT,VRA,VRB

4	VRT	VRA	VRB	8
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 7
   src1 ← EXTZ(VSR[VRA+32].byte[2×i+1])
   src2 ← EXTZ(VSR[VRB+32].byte[2×i+1])

   VSR[VRT+32].hword[i] ← CHOP16(src1 × src2)
end
```

For each integer value `i` from 0 to 7, do the following.

The unsigned integer value in byte element `i×2+1` of `VSR[VRA+32]` is multiplied by the unsigned integer value in byte element `i×2+1` of `VSR[VRB+32]`.

The 16-bit product is placed into halfword element `i` of `VSR[VRT+32]`.

Special Registers Altered:

None

Register Data Layout for vmuleub

src1	.byte[0]	unused	.byte[2]	unused	.byte[4]	unused	.byte[6]	unused	.byte[8]	unused	.byte[10]	unused	.byte[12]	unused	.byte[14]	unused
src2	.byte[0]	unused	.byte[2]	unused	.byte[4]	unused	.byte[6]	unused	.byte[8]	unused	.byte[10]	unused	.byte[12]	unused	.byte[14]	unused

result	VSR[VRT+32].hword[0]		VSR[VRT+32].hword[1]		VSR[VRT+32].hword[2]		VSR[VRT+32].hword[3]		VSR[VRT+32].hword[4]		VSR[VRT+32].hword[5]		VSR[VRT+32].hword[6]		VSR[VRT+32].hword[7]	
	0	8	16	24	32	40	48	56	64	72	80	88	96	104	112	120 127

Register Data Layout for vmuloub

src1	unused	.byte[1]	unused	.byte[3]	unused	.byte[5]	unused	.byte[7]	unused	.byte[9]	unused	.byte[11]	unused	.byte[13]	unused	.byte[15]
src2	unused	.byte[1]	unused	.byte[3]	unused	.byte[5]	unused	.byte[7]	unused	.byte[9]	unused	.byte[11]	unused	.byte[13]	unused	.byte[15]

result	VSR[VRT+32].hword[0]		VSR[VRT+32].hword[1]		VSR[VRT+32].hword[2]		VSR[VRT+32].hword[3]		VSR[VRT+32].hword[4]		VSR[VRT+32].hword[5]		VSR[VRT+32].hword[6]		VSR[VRT+32].hword[7]	
	0	8	16	24	32	40	48	56	64	72	80	88	96	104	112	120 127

Vector Multiply Even Signed Halfword VX-form

vmulesh VRT,VRA,VRB

4	VRT	VRA	VRB	840
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
   src1 ← EXTS(VSR[VRA+32].hword[2×i])
   src2 ← EXTS(VSR[VRB+32].hword[2×i])

   VSR[VRT+32].word[i] ← CHOP32(src1 × src2)
end
```

For each integer value i from 0 to 3, do the following.

The signed integer value in halfword element i×2 of VSR[VRA+32] is multiplied by the signed integer value in halfword element i×2 of VSR[VRB+32].

The 32-bit product is placed into word element i of VSR[VRT+32].

Special Registers Altered:

None

Vector Multiply Odd Signed Halfword VX-form

vmulosh VRT,VRA,VRB

4	VRT	VRA	VRB	328
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
   src1 ← EXTS(VSR[VRA+32].hword[2×i+1])
   src2 ← EXTS(VSR[VRB+32].hword[2×i+1])

   VSR[VRT+32].word[i] ← CHOP32(src1 × src2)
end
```

For each integer value i from 0 to 3, do the following.

The signed integer value in halfword element i×2+1 of VSR[VRA+32] is multiplied by the signed integer value in halfword element i×2+1 of VSR[VRB+32].

The 32-bit product is placed into word element i of VSR[VRT+32].

Special Registers Altered:

None

Register Data Layout for vmulesh

src1	VSR[VRA+32].hword[0]	unused	VSR[VRA+32].hword[2]	unused	VSR[VRA+32].hword[4]	unused	VSR[VRA+32].hword[6]	unused
src2	VSR[VRB+32].hword[0]	unused	VSR[VRB+32].hword[2]	unused	VSR[VRB+32].hword[4]	unused	VSR[VRB+32].hword[6]	unused

result	VSR[VRT+32].word[0]	VSR[VRT+32].word[1]	VSR[VRT+32].word[2]	VSR[VRT+32].word[3]
	0 16	32 48	64 80	96 112 127

Register Data Layout for vmulosh

src1	unused	VSR[VRA+32].hword[1]	unused	VSR[VRA+32].hword[3]	unused	VSR[VRA+32].hword[5]	unused	VSR[VRA+32].hword[7]
src2	unused	VSR[VRB+32].hword[1]	unused	VSR[VRB+32].hword[3]	unused	VSR[VRB+32].hword[5]	unused	VSR[VRB+32].hword[7]

result	VSR[VRT+32].word[0]	VSR[VRT+32].word[1]	VSR[VRT+32].word[2]	VSR[VRT+32].word[3]
	0 16	32 48	64 80	96 112 127

Vector Multiply Even Unsigned Halfword VX-form

vmuleuh VRT,VRA,VRB

4	VRT	VRA	VRB	584
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
   src1 ← EXTZ(VSR[VRA+32].hword[2×i])
   src2 ← EXTZ(VSR[VRB+32].hword[2×i])

   VSR[VRT+32].word[i] ← CHOP32(src1 × src2)
end
```

For each integer value i from 0 to 3, do the following.

The unsigned integer value in halfword element i×2 of VSR[VRA+32] is multiplied by the unsigned integer value in halfword element i×2 of VSR[VRB+32].

The 32-bit product is placed into word element i of VSR[VRT+32].

Special Registers Altered:

None

Vector Multiply Odd Unsigned Halfword VX-form

vmulouh VRT,VRA,VRB

4	VRT	VRA	VRB	72
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
   src1 ← EXTZ(VSR[VRA+32].hword[2×i+1])
   src2 ← EXTZ(VSR[VRB+32].hword[2×i+1])

   VSR[VRT+32].word[i] ← CHOP32(src1 × src2)
end
```

For each integer value i from 0 to 3, do the following.

The unsigned integer value in halfword element i×2+1 of VSR[VRA+32] is multiplied by the unsigned integer value in halfword element i×2+1 of VSR[VRB+32].

The 32-bit product is placed into word element i of VSR[VRT+32].

Special Registers Altered:

None

Register Data Layout for vmuleuh

src1	VSR[VRA+32].hword[0]	unused	VSR[VRA+32].hword[2]	unused	VSR[VRA+32].hword[4]	unused	VSR[VRA+32].hword[6]	unused
src2	VSR[VRB+32].hword[0]	unused	VSR[VRB+32].hword[2]	unused	VSR[VRB+32].hword[4]	unused	VSR[VRB+32].hword[6]	unused

result	VSR[VRT+32].word[0]	VSR[VRT+32].word[1]	VSR[VRT+32].word[2]	VSR[VRT+32].word[3]
	0 16	32 48	64 80	96 112 127

Register Data Layout for vmulouh

src1	unused	VSR[VRA+32].hword[1]	unused	VSR[VRA+32].hword[3]	unused	VSR[VRA+32].hword[5]	unused	VSR[VRA+32].hword[7]
src2	unused	VSR[VRB+32].hword[1]	unused	VSR[VRB+32].hword[3]	unused	VSR[VRB+32].hword[5]	unused	VSR[VRB+32].hword[7]

result	VSR[VRT+32].word[0]	VSR[VRT+32].word[1]	VSR[VRT+32].word[2]	VSR[VRT+32].word[3]
	0 16	32 48	64 80	96 112 127

Vector Multiply Even Signed Word VX-form

vmulesw VRT,VRA,VRB

4	VRT	VRA	VRB	904
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 1
   src1 ← EXTS(VSR[VRA+32].word[2×i])
   src2 ← EXTS(VSR[VRB+32].word[2×i])

   VSR[VRT+32].dword[i] ← CHOP64(src1 × src2)
end
```

For each integer value i from 0 to 1, do the following.

The signed integer in word element 2×i of VSR[VRA+32] is multiplied by the signed integer in word element 2×i of VSR[VRB+32].

The 64-bit product is placed into doubleword element i of VSR[VRT+32].

Special Registers Altered:

None

Vector Multiply Odd Signed Word VX-form

vmulosw VRT,VRA,VRB

4	VRT	VRA	VRB	392
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 1
   src1 ← EXTS(VSR[VRA+32].word[2×i+1])
   src2 ← EXTS(VSR[VRB+32].word[2×i+1])

   VSR[VRT+32].dword[i] ← CHOP64(src1 × src2)
end
```

For each integer value i from 0 to 1, do the following.

The signed integer in word element 2×i+1 of VSR[VRA+32] is multiplied by the signed integer in word element 2×i+1 of VSR[VRB+32].

The 64-bit product is placed into doubleword element i of VSR[VRT+32].

Special Registers Altered:

None

Register Data Layout for vmulesw

src1	VSR[VRA+32].word[0]	unused	VSR[VRA+32].word[2]	unused
src2	VSR[VRB+32].word[0]	unused	VSR[VRB+32].word[2]	unused

result	VSR[VRT+32].dword[0]		VSR[VRT+32].dword[1]	
	0 32		64 96	127

Register Data Layout for vmulosw

src1	unused	VSR[VRA+32].word[1]	unused	VSR[VRA+32].word[3]
src2	unused	VSR[VRB+32].word[1]	unused	VSR[VRB+32].word[3]

result	VSR[VRT+32].dword[0]		VSR[VRT+32].dword[1]	
	0 32		64 96	127

Vector Multiply Even Unsigned Word VX-form

vmuleuw VRT,VRA,VRB

4	VRT	VRA	VRB	648
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 1
   src1 ← EXTZ(VSR[VRA+32].word[2×i])
   src2 ← EXTZ(VSR[VRB+32].word[2×i])

   VSR[VRT+32].dword[i] ← CHOP64(src1 × src2)
end
```

For each integer value i from 0 to 1, do the following.

The unsigned integer in word element 2×i of VSR[VRA+32] is multiplied by the unsigned integer in word element 2×i of VSR[VRB+32].

The 64-bit product is placed into doubleword element i of VSR[VRT+32].

Special Registers Altered:

None

Vector Multiply Odd Unsigned Word VX-form

vmulouw VRT,VRA,VRB

4	VRT	VRA	VRB	136
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 1
   src1 ← EXTZ(VSR[VRA+32].word[2×i+1])
   src2 ← EXTZ(VSR[VRB+32].word[2×i+1])

   VSR[VRT+32].dword[i] ← CHOP64(src1 × src2)
end
```

For each integer value i from 0 to 1, do the following.

The unsigned integer in word element 2×i+1 of VSR[VRA+32] is multiplied by the unsigned integer in word element 2×i+1 of VSR[VRB+32].

The 64-bit product is placed into doubleword element i of VSR[VRT+32].

Special Registers Altered:

None

Register Data Layout for vmuleuw

src1	VSR[VRA+32].word[0]	unused	VSR[VRA+32].word[2]	unused
src2	VSR[VRB+32].word[0]	unused	VSR[VRB+32].word[2]	unused

result	VSR[VRT+32].dword[0]		VSR[VRT+32].dword[1]	
	0 32		64 96	127

Register Data Layout for vmulouw

src1	unused	VSR[VRA+32].word[1]	unused	VSR[VRA+32].word[3]
src2	unused	VSR[VRB+32].word[1]	unused	VSR[VRB+32].word[3]

result	VSR[VRT+32].dword[0]		VSR[VRT+32].dword[1]	
	0 32		64 96	127

Vector Multiply Even Unsigned Doubleword VX-form

vmuleud VRT,VRA,VRB

4	VRT	VRA	VRB	712
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

src1 ← EXTZ(VSR[VRA+32].dword[0])
src2 ← EXTZ(VSR[VRB+32].dword[0])

VSR[VRT+32] ← CHOP128(src1 × src2)
```

Let src1 be the unsigned integer value in doubleword element 0 of VSR[VRA+32].

Let src2 be the unsigned integer value in doubleword element 0 of VSR[VRB+32].

The 128-bit product of src1 multiplied by src2 is placed into VSR[VRT+32].

Special Registers Altered:

None

Vector Multiply Odd Unsigned Doubleword VX-form

vmuloud VRT,VRA,VRB

4	VRT	VRA	VRB	200
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

src1 ← EXTZ(VSR[VRA+32].dword[1])
src2 ← EXTZ(VSR[VRB+32].dword[1])

VSR[VRT+32] ← CHOP128(src1 × src2)
```

Let src1 be the unsigned integer value in doubleword element 1 of VSR[VRA+32].

Let src2 be the unsigned integer value in doubleword element 1 of VSR[VRB+32].

The 128-bit product of src1 multiplied by src2 is placed into VSR[VRT+32].

Special Registers Altered:

None

Register Data Layout for vmuleud

src1	VSR[VRA+32].dword[0]	unused
src2	VSR[VRB+32].dword[0]	unused

result	VSR[VRT+32]

0 64 127

Register Data Layout for vmuloud

src1	unused	VSR[VRA+32].dword[1]
src2	unused	VSR[VRB+32].dword[1]

result	VSR[VRT+32]

0 64 127

Vector Multiply Even Signed Doubleword VX-form

vmulesd VRT,VRA,VRB

4	VRT	VRA	VRB	968
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

src1 ← EXTS(VSR[VRA+32].dword[0])
src2 ← EXTS(VSR[VRB+32].dword[0])

VSR[VRT+32] ← CHOP128(src1 × src2)
```

Let src1 be the signed integer value in doubleword element 0 of VSR[VRA+32].

Let src2 be the signed integer value in doubleword element 0 of VSR[VRB+32].

The 128-bit product of src1 multiplied by src2 is placed into VSR[VRT+32].

Special Registers Altered:

None

Vector Multiply Odd Signed Doubleword VX-form

vmulosd VRT,VRA,VRB

4	VRT	VRA	VRB	456
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

src1 ← EXTS(VSR[VRA+32].dword[1])
src2 ← EXTS(VSR[VRB+32].dword[1])

VSR[VRT+32] ← CHOP128(src1 × src2)
```

Let src1 be the signed integer value in doubleword element 1 of VSR[VRA+32].

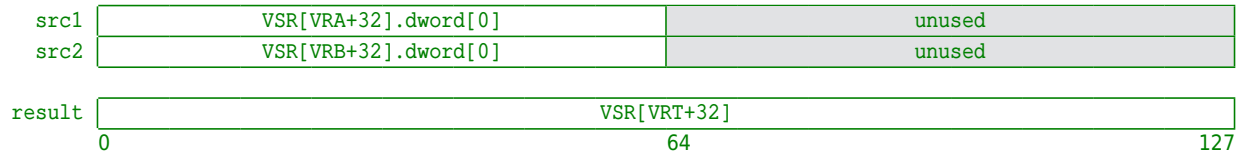
Let src2 be the signed integer value in doubleword element 1 of VSR[VRB+32].

The 128-bit product of src1 multiplied by src2 is placed into VSR[VRT+32].

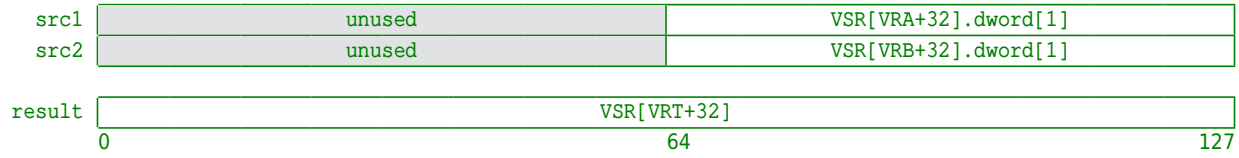
Special Registers Altered:

None

Register Data Layout for vmulesd



Register Data Layout for vmulosd



Vector Multiply Unsigned Word Modulo VX-form

vmuluwm VRT,VRA,VRB

4	VRT	VRA	VRB	137
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
   src1 ← EXTZ(VSR[VRA+32].word[i])
   src2 ← EXTZ(VSR[VRB+32].word[i])

   VSR[VRT+32].word[i] ← CHOP32(src1 × src2)
end
```

For each integer value `i` from 0 to 3, do the following.

The integer in word element `i` of `VSR[VRA+32]` is multiplied by the integer in word element `i` of `VSR[VRB+32]`.

The low-order 32 bits of the product are placed into word element `i` of `VSR[VRT+32]`.

Special Registers Altered:

None

Programming Note

vmuluwm can be used for unsigned or signed integers.

Register Data Layout for vmuluwm

src1	VSR[VRA+32].word[0]	VSR[VRA+32].word[1]	VSR[VRA+32].word[2]	VSR[VRA+32].word[3]
src2	VSR[VRB+32].word[0]	VSR[VRB+32].word[1]	VSR[VRB+32].word[2]	VSR[VRB+32].word[3]

result	VSR[VRT+32].word[0]	VSR[VRT+32].word[1]	VSR[VRT+32].word[2]	VSR[VRT+32].word[3]
	0	32	64	96 127

Vector Multiply High Signed Word VX-form

vmulhsw VRT,VRA,VRB

4	VRT	VRA	VRB	905
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
   src1 ← EXTS(VSR[VRA+32].word[i])
   src2 ← EXTS(VSR[VRB+32].word[i])

   VSR[VRT+32].word[i] ← CHOP32((src1 × src2) >> 32)
end
```

For each integer value `i` from 0 to 3, do the following.

The signed integer value in word element `i` of `VSR[VRA+32]` is multiplied by the signed integer value in word element `i` of `VSR[VRB+32]`.

The high-order 32 bits of the 64-bit product are placed into word element `i` of `VSR[VRT+32]`.

Special Registers Altered:

None

Vector Multiply High Unsigned Word VX-form

vmulhuw VRT,VRA,VRB

4	VRT	VRA	VRB	649
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
   src1 ← EXTZ(VSR[VRA+32].word[i])
   src2 ← EXTZ(VSR[VRB+32].word[i])

   VSR[VRT+32].word[i] ← CHOP32((src1 × src2) >> 32)
end
```

For each integer value `i` from 0 to 3, do the following.

The unsigned integer value in word element `i` of `VSR[VRA+32]` is multiplied by the unsigned integer value in word element `i` of `VSR[VRB+32]`.

The high-order 32 bits of the 64-bit product are placed into word element `i` of `VSR[VRT+32]`.

Special Registers Altered:

None

Register Data Layout for vmulhsw

src1	VSR[VRA+32].word[0]	VSR[VRA+32].word[1]	VSR[VRA+32].word[2]	VSR[VRA+32].word[3]
src2	VSR[VRB+32].word[0]	VSR[VRB+32].word[1]	VSR[VRB+32].word[2]	VSR[VRB+32].word[3]
result	VSR[VRT+32].word[0]	VSR[VRT+32].word[1]	VSR[VRT+32].word[2]	VSR[VRT+32].word[3]
	0	32	64	96 127

Register Data Layout for vmulhuw

src1	VSR[VRA+32].word[0]	VSR[VRA+32].word[1]	VSR[VRA+32].word[2]	VSR[VRA+32].word[3]
src2	VSR[VRB+32].word[0]	VSR[VRB+32].word[1]	VSR[VRB+32].word[2]	VSR[VRB+32].word[3]
result	VSR[VRT+32].word[0]	VSR[VRT+32].word[1]	VSR[VRT+32].word[2]	VSR[VRT+32].word[3]
	0	32	64	96 127

Vector Multiply High Signed Doubleword VX-form

vmulhsd VRT,VRA,VRB

4	VRT	VRA	VRB	969
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 1
   src1 ← EXTS(VSR[VRA+32].dword[i])
   src2 ← EXTS(VSR[VRB+32].dword[i])

   VSR[VRT+32].dword[i] ← CHOP64((src1 × src2) >> 64)
end
```

For each integer value `i` from 0 to 1, do the following.

The signed integer value in doubleword element `i` of `VSR[VRA+32]` is multiplied by the signed integer value in doubleword element `i` of `VSR[VRB+32]`.

The high-order 64 bits of the 128-bit product are placed into doubleword element `i` of `VSR[VRT+32]`.

Special Registers Altered:

None

Vector Multiply High Unsigned Doubleword VX-form

vmulhud VRT,VRA,VRB

4	VRT	VRA	VRB	713
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 1
   src1 ← EXTZ(VSR[VRA+32].dword[i])
   src2 ← EXTZ(VSR[VRB+32].dword[i])

   VSR[VRT+32].dword[i] ← CHOP64((src1 × src2) >> 64)
end
```

For each integer value `i` from 0 to 1, do the following.

The unsigned integer value in doubleword element `i` of `VSR[VRA+32]` is multiplied by the unsigned integer value in doubleword element `i` of `VSR[VRB+32]`.

The high-order 64 bits of the 128-bit product are placed into doubleword element `i` of `VSR[VRT+32]`.

Special Registers Altered:

None

Register Data Layout for vmulhsd

src1	VSR[VRA+32].dword[0]	VSR[VRA+32].dword[1]
src2	VSR[VRB+32].dword[0]	VSR[VRB+32].dword[1]
result	VSR[VRT+32].dword[0]	VSR[VRT+32].dword[1]
	0 64	127

Register Data Layout for vmulhud

src1	VSR[VRA+32].dword[0]	VSR[VRA+32].dword[1]
src2	VSR[VRB+32].dword[0]	VSR[VRB+32].dword[1]
result	VSR[VRT+32].dword[0]	VSR[VRT+32].dword[1]
	0 64	127

Vector Multiply Low Doubleword VX-form

vmulld VRT,VRA,VRB

4	VRT	VRA	VRB	457
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 1
   src1 ← EXTS(VSR[VRA+32].dword[i])
   src2 ← EXTS(VSR[VRB+32].dword[i])

   VSR[VRT+32].dword[i] ← CHOP64(src1 × src2)
end
```

For each integer value `i` from 0 to 1, do the following.

The integer value in doubleword element `i` of `VSR[VRA+32]` is mutiplied by the integer value in doubleword element `i` of `VSR[VRB+32]`.

The low-order 64 bits of the product are placed into doubleword element `i` of `VSR[VRT+32]`.

Special Registers Altered:

None

Register Data Layout for vmulld

src1	VSR[VRA+32].dword[0]	VSR[VRA+32].dword[1]
src2	VSR[VRB+32].dword[0]	VSR[VRB+32].dword[1]

result	VSR[VRT+32].dword[0]	VSR[VRT+32].dword[1]
	0 64	127

6.9.1.4 Vector Integer Multiply-Add/Sum Instructions

Vector Multiply-High-Add Signed Halfword Saturate VA-form

vmhaddshs VRT,VRA,VRB,VRC

4	VRT	VRA	VRB	VRC	32
0	6	11	16	21	26 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 7
   src1 ← EXTS(VSR[VRA+32].hword[i])
   src2 ← EXTS(VSR[VRB+32].hword[i])
   src3 ← EXTS(VSR[VRC+32].hword[i])

   result ← ((src1 × src2) >> 15) + src3
   VSR[VRT+32].hword[i] ← si16_CLAMP(result)
   VSCR.SAT ← sat_flag
end
```

For each integer value `i` from 0 to 7, do the following.

The signed integer value in halfword element `i` of `VSR[VRA+32]` is multiplied by the signed integer value in halfword element `i` of `VSR[VRB+32]`, producing a 32-bit signed integer product.

Bits 0:16 of the product are added to the signed integer value in halfword element `i` of `VSR[VRC+32]`.

- If the intermediate result is greater than $2^{15}-1$, the result saturates to $2^{15}-1$ and `SAT` is set to 1.
- If the intermediate result is less than -2^{15}, the result saturates to -2^{15} and `SAT` is set to 1.

The low-order 16 bits of the result are placed into halfword element `i` of `VSR[VRT+32]`.

Special Registers Altered:

Register	*Field(s)*
VSCR	SAT

Vector Multiply-High-Round-Add Signed Halfword Saturate VA-form

vmhraddshs VRT,VRA,VRB,VRC

4	VRT	VRA	VRB	VRC	33
0	6	11	16	21	26 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 7
   src1 ← EXTS(VSR[VRA+32].hword[i])
   src2 ← EXTS(VSR[VRB+32].hword[i])
   src3 ← EXTS(VSR[VRC+32].hword[i])

   result ← src3 +
      (((src1 × src2) + 0x0000_4000) >> 15)
   VSR[VRT+32].hword[i] ← si16_CLAMP(result)
   VSCR.SAT ← sat_flag
end
```

For each integer value `i` from 0 to 7, do the following.

The signed integer value in halfword element `i` of `VSR[VRA+32]` is multiplied by the signed integer value in halfword element `i` of `VSR[VRB+32]`, producing a 32-bit signed integer product.

The value `0x0000_4000` is added to the product.

Bits 0:16 of the 32-bit sum are added to the signed integer value in halfword element `i` of `VSR[VRC+32]`.

- If the intermediate result is greater than $2^{15}-1$, the result saturates to $2^{15}-1$ and `SAT` is set to 1.
- If the intermediate result is less than -2^{15}, the result saturates to -2^{15} and `SAT` is set to 1.

The low-order 16 bits of the result are placed into halfword element `i` of `VSR[VRT+32]`.

Special Registers Altered:

Register	*Field(s)*
VSCR	SAT

Register Data Layout for vmhaddshs & vmhraddshs

src1	VSR[VRA+32].hword[0]	VSR[VRA+32].hword[1]	VSR[VRA+32].hword[2]	VSR[VRA+32].hword[3]	VSR[VRA+32].hword[4]	VSR[VRA+32].hword[5]	VSR[VRA+32].hword[6]	VSR[VRA+32].hword[7]
src2	VSR[VRB+32].hword[0]	VSR[VRB+32].hword[1]	VSR[VRB+32].hword[2]	VSR[VRB+32].hword[3]	VSR[VRB+32].hword[4]	VSR[VRB+32].hword[5]	VSR[VRB+32].hword[6]	VSR[VRB+32].hword[7]
result	VSR[VRT+32].hword[0]	VSR[VRT+32].hword[1]	VSR[VRT+32].hword[2]	VSR[VRT+32].hword[3]	VSR[VRT+32].hword[4]	VSR[VRT+32].hword[5]	VSR[VRT+32].hword[6]	VSR[VRT+32].hword[7]
	0	16	32	48	64	80	96	112 127

Vector Multiply-Low-Add Unsigned Halfword Modulo VA-form

vmladduhm VRT,VRA,VRB,VRC

4	VRT	VRA	VRB	VRC	34
0	6	11	16	21	26 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 7
   src1 ← EXTZ(VSR[VRA+32].hword[i])
   src2 ← EXTZ(VSR[VRB+32].hword[i])
   src3 ← EXTZ(VSR[VRC+32].hword[i])

   VSR[VRT+32].hword[i] ←
      CHOP16((src1 × src2) + src3)
end
```

For each integer value i from 0 to 7, do the following.

The unsigned integer value in halfword element i of VSR[VRA+32] is multiplied by the unsigned integer value in halfword element i in VSR[VRB+32].

The product is added to the unsigned integer value in halfword element i of VSR[VRC+32].

The low-order 16 bits of the sum of the product and the unsigned integer value in word element i of VSR[VRC+32] are placed into word element i of VSR[VRT+32].

Special Registers Altered:

None

Programming Note

vmladduhm can be used for unsigned or signed integers.

Vector Multiply-Sum Unsigned Byte Modulo VA-form

vmsumubm VRT,VRA,VRB,VRC

4	VRT	VRA	VRB	VRC	36
0	6	11	16	21	26 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
   temp ← EXTZ(VSR[VRC+32].word[i])
   do j = 0 to 3
      src1 ← EXTZ(VSR[VRA+32].word[i].byte[j])
      src2 ← EXTZ(VSR[VRB+32].word[i].byte[j])
      temp ← temp + (src1 × src2)
   end
   VSR[VRT+32].word[i] ← CHOP32(temp)
end
```

For each integer value i from 0 to 3, do the following.

For each integer value j from 0 to 3, do the following.

The unsigned integer value in byte element j of word element i of VSR[VRA+32] is multiplied by the unsigned integer value in byte element j of word element i of VSR[VRB+32].

The sum of the four products is added to the unsigned integer value in word element i of VSR[VRC+32].

The low-order 32 bits of the result are placed into word element i of VSR[VRT+32].

Special Registers Altered:

None

Register Data Layout for vmladduhm

src1	VSR[VRA+32].hword[0]	VSR[VRA+32].hword[1]	VSR[VRA+32].hword[2]	VSR[VRA+32].hword[3]	VSR[VRA+32].hword[4]	VSR[VRA+32].hword[5]	VSR[VRA+32].hword[6]	VSR[VRA+32].hword[7]
src2	VSR[VRB+32].hword[0]	VSR[VRB+32].hword[1]	VSR[VRB+32].hword[2]	VSR[VRB+32].hword[3]	VSR[VRB+32].hword[4]	VSR[VRB+32].hword[5]	VSR[VRB+32].hword[6]	VSR[VRB+32].hword[7]
src3	VSR[VRC+32].hword[0]	VSR[VRC+32].hword[1]	VSR[VRC+32].hword[2]	VSR[VRC+32].hword[3]	VSR[VRC+32].hword[4]	VSR[VRC+32].hword[5]	VSR[VRC+32].hword[6]	VSR[VRC+32].hword[7]
result	VSR[VRT+32].hword[0]	VSR[VRT+32].hword[1]	VSR[VRT+32].hword[2]	VSR[VRT+32].hword[3]	VSR[VRT+32].hword[4]	VSR[VRT+32].hword[5]	VSR[VRT+32].hword[6]	VSR[VRT+32].hword[7]
	0	16	32	48	64	80	96	112 127

Register Data Layout for vmsumubm

src1	.byte[0]	.byte[1]	.byte[2]	.byte[3]	.byte[4]	.byte[5]	.byte[6]	.byte[7]	.byte[8]	.byte[9]	.byte[10]	.byte[11]	.byte[12]	.byte[13]	.byte[14]	.byte[15]
src2	.byte[0]	.byte[1]	.byte[2]	.byte[3]	.byte[4]	.byte[5]	.byte[6]	.byte[7]	.byte[8]	.byte[9]	.byte[10]	.byte[11]	.byte[12]	.byte[13]	.byte[14]	.byte[15]
src3	VSR[VRC+32].word[0]				VSR[VRC+32].word[1]				VSR[VRC+32].word[2]				VSR[VRC+32].word[3]			
result	VSR[VRT+32].word[0]				VSR[VRT+32].word[1]				VSR[VRT+32].word[2]				VSR[VRT+32].word[3]			
	0	8	16	24	32	40	48	56	64	72	80	88	96	104	112	120 127

Vector Multiply-Sum Mixed Byte Modulo VA-form

vmsummbm VRT,VRA,VRB,VRC

4	VRT	VRA	VRB	VRC	37
0	6	11	16	21	26 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
   temp ← EXTS(VSR[VRC+32].word[i])
   do j = 0 to 3
      src1 ← EXTS(VSR[VRA+32].word[i].byte[j])
      src2 ← EXTZ(VSR[VRB+32].word[i].byte[j])
      temp ← temp + (src1 × src2)
   end
   VSR[VRT+32].word[i] ← CHOP32(temp)
end
```

For each integer value `i` from 0 to 3, do the following.

For each integer value `j` from 0 to 3, do the following.

The signed integer value in byte element `j` of word element `i` of `VSR[VRA+32]` is multiplied by the unsigned integer value in byte element `j` of word element `i` of `VSR[VRB+32]`.

The sum of the four products is added to the signed integer value in word element `i` of `VSR[VRC+32]`.

The low-order 32 bits of the result are placed into word element `i` of `VSR[VRT+32]`.

Special Registers Altered:

None

Vector Multiply-Sum Signed Halfword Modulo VA-form

vmsumshm VRT,VRA,VRB,VRC

4	VRT	VRA	VRB	VRC	40
0	6	11	16	21	26 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
   temp ← EXTS(VSR[VRC+32].word[i])
   do j = 0 to 1
      src1 ← EXTS(VSR[VRA+32].word[i].hword[j])
      src2 ← EXTS(VSR[VRB+32].word[i].hword[j])
      temp ← temp + (src1 × src2)
   end
   VSR[VRT+32].word[i] ← CHOP32(temp)
end
```

For each integer value `i` from 0 to 3, do the following.

For each integer value `j` from 0 to 1, do the following.

The signed integer value in halfword element `j` of word element `i` of `VSR[VRA+32]` is multiplied by the signed integer value in halfword element `j` of word element `i` of `VSR[VRB+32]`.

The sum of the two products is added to the signed integer value in word element `i` of `VSR[VRC+32]`.

The low-order 32 bits of the result are placed into word element `i` of `VSR[VRT+32]`.

Special Registers Altered:

None

Register Data Layout for vmsummbm

src1	.byte[0]	.byte[1]	.byte[2]	.byte[3]	.byte[4]	.byte[5]	.byte[6]	.byte[7]	.byte[8]	.byte[9]	.byte[10]	.byte[11]	.byte[12]	.byte[13]	.byte[14]	.byte[15]
src2	.byte[0]	.byte[1]	.byte[2]	.byte[3]	.byte[4]	.byte[5]	.byte[6]	.byte[7]	.byte[8]	.byte[9]	.byte[10]	.byte[11]	.byte[12]	.byte[13]	.byte[14]	.byte[15]
src3	VSR[VRC+32].word[0]				VSR[VRC+32].word[1]				VSR[VRC+32].word[2]				VSR[VRC+32].word[3]			
result	VSR[VRT+32].word[0]				VSR[VRT+32].word[1]				VSR[VRT+32].word[2]				VSR[VRT+32].word[3]			
	0	8	16	24	32	40	48	56	64	72	80	88	96	104	112	120 127

Register Data Layout for vmsumshm

src1	VSR[VRA+32].hword[0]	VSR[VRA+32].hword[1]	VSR[VRA+32].hword[2]	VSR[VRA+32].hword[3]	VSR[VRA+32].hword[4]	VSR[VRA+32].hword[5]	VSR[VRA+32].hword[6]	VSR[VRA+32].hword[7]
src2	VSR[VRB+32].hword[0]	VSR[VRB+32].hword[1]	VSR[VRB+32].hword[2]	VSR[VRB+32].hword[3]	VSR[VRB+32].hword[4]	VSR[VRB+32].hword[5]	VSR[VRB+32].hword[6]	VSR[VRB+32].hword[7]
src3	VSR[VRC+32].word[0]		VSR[VRC+32].word[1]		VSR[VRC+32].word[2]		VSR[VRC+32].word[3]	
result	VSR[VRT+32].word[0]		VSR[VRT+32].word[1]		VSR[VRT+32].word[2]		VSR[VRT+32].word[3]	
	0	16	32	48	64	80	96	112 127

Vector Multiply-Sum Signed Halfword Saturate VA-form

vmsumshs VRT,VRA,VRB,VRC

4	VRT	VRA	VRB	VRC	41
0	6	11	16	21	26 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
   temp ← EXTS(VSR[VRC+32].word[i])
   do j = 0 to 1
      src1 ← EXTS(VSR[VRA+32].word[i].hword[j])
      src2 ← EXTS(VSR[VRB+32].word[i].hword[j])
      temp ← temp + (src1 × src2)
   end
   VSR[VRT+32].word[i] ← si32_CLAMP(temp)
   VSCR.SAT ← sat_flag
end
```

For each integer value i from 0 to 3, do the following.

For each integer value j from 0 to 1, do the following.

The signed integer value in halfword element j of word element i of VSR[VRA+32] is multiplied by the signed integer value in halfword element j of word element i of VSR[VRB+32].

The sum of the two products is added to the signed integer value in word element i of VSR[VRC+32].

- If the intermediate result is greater than $2^{31}-1$, the result saturates to $2^{31}-1$ and SAT is set to 1
- If the intermediate result is less than -2^{31}, it saturates to -2^{31} and SAT is set to 1

The low-order 32 bits of the result are placed into word element i of VSR[VRT+32].

Special Registers Altered:

Register	*Field(s)*
VSCR	SAT

Vector Multiply-Sum Unsigned Halfword Modulo VA-form

vmsumuhm VRT,VRA,VRB,VRC

4	VRT	VRA	VRB	VRC	38
0	6	11	16	21	26 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
   temp ← EXTZ(VSR[VRC+32].word[i])
   do j = 0 to 1
      src1 ← EXTZ(VSR[VRA+32].word[i].hword[j])
      src2 ← EXTZ(VSR[VRB+32].word[i].hword[j])
      temp ← temp + (src1 × src2)
   end
   VSR[VRT+32].word[i] ← CHOP32(temp)
end
```

For each integer value i from 0 to 3, do the following.

For each integer value j from 0 to 1, do the following.

The unsigned integer value in halfword element j of word element i of VSR[VRA+32] is multiplied by the unsigned integer value in halfword element j of word element i of VSR[VRB+32].

The sum of the two products is added to the signed integer value in word element i of VSR[VRC+32].

The low-order 32 bits of the result are placed into word element i of VSR[VRT+32].

Special Registers Altered:

None

Register Data Layout for vmsumshs & vmsumuhm

src1	VSR[VRA+32].hword[0]	VSR[VRA+32].hword[1]	VSR[VRA+32].hword[2]	VSR[VRA+32].hword[3]	VSR[VRA+32].hword[4]	VSR[VRA+32].hword[5]	VSR[VRA+32].hword[6]	VSR[VRA+32].hword[7]
src2	VSR[VRB+32].hword[0]	VSR[VRB+32].hword[1]	VSR[VRB+32].hword[2]	VSR[VRB+32].hword[3]	VSR[VRB+32].hword[4]	VSR[VRB+32].hword[5]	VSR[VRB+32].hword[6]	VSR[VRB+32].hword[7]
src3	VSR[VRC+32].word[0]		VSR[VRC+32].word[1]		VSR[VRC+32].word[2]		VSR[VRC+32].word[3]	

result	VSR[VRT+32].word[0]	VSR[VRT+32].word[1]	VSR[VRT+32].word[2]	VSR[VRT+32].word[3]
	0 16	32 48	64 80	96 112 127

Vector Multiply-Sum Unsigned Halfword Saturate VA-form

vmsumuhs VRT,VRA,VRB,VRC

4	VRT	VRA	VRB	VRC	39
0	6	11	16	21	26 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
   temp ← EXTZ(VSR[VRC+32].word[i])
   do j = 0 to 1
      src1 ← EXTZ(VSR[VRA+32].word[i].hword[j])
      src2 ← EXTZ(VSR[VRB+32].word[i].hword[j])
      temp ← temp + src1 × src2
   end
   VSR[VRT+32].word[i] ← ui32_CLAMP(temp)
   VSCR.SAT ← sat_flag
end
```

For each integer value `i` from 0 to 3, do the following.

For each integer value `j` from 0 to 1, do the following.

The unsigned integer value in halfword element `j` of word element `i` of `VSR[VRA+32]` is multiplied by the unsigned integer value in halfword element `j` of word element `i` of `VSR[VRB+32]`.

The sum of the two products is added to the signed integer value in word element `i` of `VSR[VRC+32]`.

- If the intermediate result is greater than $2^{32}-1$, the result saturates to $2^{32}-1$ and `SAT` is set to 1
- If the intermediate result is less than -2^{32}, it saturates to -2^{32}, and `SAT` is set to 1

The low-order 32 bits of the result are placed into word element `i` of `VSR[VRT+32]`.

Special Registers Altered:

Register	*Field(s)*
VSCR	SAT

Register Data Layout for vmsumuhs

src1	VSR[VRA+32].hword[0]	VSR[VRA+32].hword[1]	VSR[VRA+32].hword[2]	VSR[VRA+32].hword[3]	VSR[VRA+32].hword[4]	VSR[VRA+32].hword[5]	VSR[VRA+32].hword[6]	VSR[VRA+32].hword[7]	
src2	VSR[VRB+32].hword[0]	VSR[VRB+32].hword[1]	VSR[VRB+32].hword[2]	VSR[VRB+32].hword[3]	VSR[VRB+32].hword[4]	VSR[VRB+32].hword[5]	VSR[VRB+32].hword[6]	VSR[VRB+32].hword[7]	
src3	VSR[VRC+32].word[0]		VSR[VRC+32].word[1]		VSR[VRC+32].word[2]		VSR[VRC+32].word[3]		
result	VSR[VRT+32].word[0]		VSR[VRT+32].word[1]		VSR[VRT+32].word[2]		VSR[VRT+32].word[3]		
	0	16	32	48	64	80	96	112	127

Vector Multiply-Sum Unsigned Doubleword Modulo VA-form

vmsumudm VRT,VRA,VRB,VRC

4	VRT	VRA	VRB	VRC	35
0	6	11	16	21	26 31

```
if MSR.VEC=0 then Vector_Unavailable()

temp ← EXTZ(VSR[VRC+32])
do i = 0 to 1
   src1  ← EXTZ(VSR[VRA+32].dword[i])
   src2  ← EXTZ(VSR[VRB+32].dword[i])
   temp ← temp + (src1 × src2)
end
VSR[VRT+32] ← CHOP128(temp)
```

Let prod0 be the product of the unsigned integer values in doubleword element 0 of VSR[VRA+32] and doubleword element 0 of VSR[VRB+32].

Let prod1 be the product of the unsigned integer values in doubleword element 1 of VSR[VRA+32] and doubleword element 1 of VSR[VRB+32].

The low-order 128 bits of the sum of prod0, prod1, and the unsigned integer value in VSR[VRC+32] are placed into VSR[VRT+32].

Special Registers Altered:

None

Programming Note

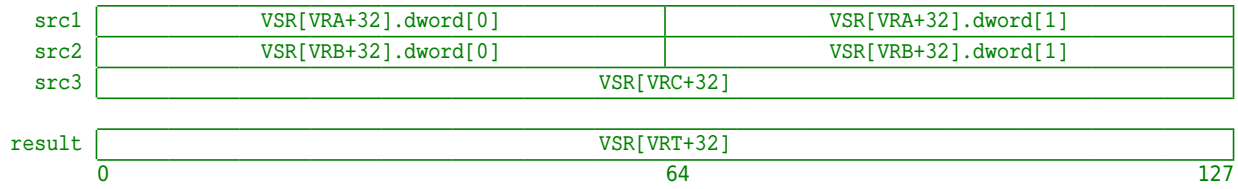
A horizontal add of the doubleword elements in VSR[VRA+32] can be performed using *vmsumudm* when VSR[VRB+32] contains the doubleword integer values {1,1} and VSR[VRC+32] contains the quadword integer value 0.

A horizontal subtract of the doubleword elements in VSR[VRA+32] can be performed using *vmsumudm* when VSR[VRB+32] contains the doubleword integer values {1,−1} and VSR[VRC+32] 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 VSR[VRA+32] or VSR[VRB+32] are 0 and the contents of VSR[VRC+32] to 0.

A multiply odd unsigned doubleword operation can be performed using *vmsumudm* when the contents of doubleword element 0 of VSR[VRA+32] or VSR[VRB+32] are 0 and the contents of VSR[VRC+32] to 0.

Register Data Layout for vmsumudm



Vector Multiply-Sum & write Carry-out Unsigned Doubleword VA-form

vmsumcud VRT,VRA,VRB,VRC

4	VRT	VRA	VRB	VRC	23
0	6	11	16	21	26 31

```
if MSR.VEC=0 then Vector_Unavailable()

temp ← EXTZ(VSR[VRC+32])
do i = 0 to 1
   src1  ← EXTZ(VSR[VRA+32].dword[i])
   src2  ← EXTZ(VSR[VRB+32].dword[i])
   temp ← temp + (src1 × src2)
end

VSR[VRT+32] ← CHOP128(temp >> 128)
```

Let prod0 be the quadword product of the unsigned integer values in doubleword element 0 of VSR[VRA+32] and doubleword element 0 of VSR[VRB+32].

Let prod1 be the quadword product of the unsigned integer values in doubleword element 1 of VSR[VRA+32] and doubleword element 1 of VSR[VRB+32].

The carry out of the low-order 128 bits of the sum of prod0, prod1, and the unsigned integer value in VSR[VRC+32] is placed into VSR[VRT+32].

Special Registers Altered:

None

Register Data Layout for vmsumcud

src1	VSR[VRA+32].dword[0]	VSR[VRA+32].dword[1]
src2	VSR[VRB+32].dword[0]	VSR[VRB+32].dword[1]
src3	VSR[VRC+32]	

result	VSR[VRT+32]

0 64 127

6.9.1.5 Vector Integer Divide Instructions

Vector Divide Signed Word VX-form

vdivsw VRT,VRA,VRB

4	VRT	VRA	VRB	395
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
   dividend ← EXTS(VSR[VRA+32].word[i])
   divisor  ← EXTS(VSR[VRB+32].word[i])

   VSR[VRT+32].word[i] ← CHOP32(dividend ÷ divisor)
end
```

For each integer value `i` from 0 to 3, do the following.

The signed integer in word element `i` of `VSR[VRA+32]` is divided by the signed integer in word element `i` of `VSR[VRB+32]`.

The quotient is the unique signed integer that satisfies

$$\text{dividend} = (\text{quotient} \times \text{divisor}) + \text{remainder}$$

where $0 \leq \text{remainder} < |\text{divisor}|$ if the dividend is nonnegative, and $-|\text{divisor}| < \text{remainder} \leq 0$ if the dividend is negative.

If an attempt is made to perform any of the divisions

$$0x8000\_0000 \div -1$$
$$\langle \text{anything} \rangle \div 0$$

then the quotient is undefined.

The quotient is placed into word element `i` of `VSR[VRT+32]`.

Special Registers Altered:

None

Vector Divide Unsigned Word VX-form

vdivuw VRT,VRA,VRB

4	VRT	VRA	VRB	139
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
   dividend ← EXTZ(VSR[VRA+32].word[i])
   divisor  ← EXTZ(VSR[VRB+32].word[i])

   VSR[VRT+32].word[i] ← CHOP32(dividend ÷ divisor)
end
```

For each integer value `i` from 0 to 3, do the following.

The unsigned integer in word element `i` of `VSR[VRA+32]` is divided by the unsigned integer in word element `i` of `VSR[VRB+32]`.

The quotient is the unique unsigned integer that satisfies

$$\text{dividend} = (\text{quotient} \times \text{divisor}) + \text{remainder}$$

where $0 \leq \text{remainder} < |\text{divisor}|$.

If an attempt is made to perform the division

$$\langle \text{anything} \rangle \div 0$$

then the quotient is undefined.

The quotient is placed into word element `i` of `VSR[VRT+32]`.

Special Registers Altered:

None

Register Data Layout for vdivsw & vdivuw

src1	VSR[VRA+32].word[0]	VSR[VRA+32].word[1]	VSR[VRA+32].word[2]	VSR[VRA+32].word[3]
src2	VSR[VRB+32].word[0]	VSR[VRB+32].word[1]	VSR[VRB+32].word[2]	VSR[VRB+32].word[3]

result	VSR[VRT+32].word[0]	VSR[VRT+32].word[1]	VSR[VRT+32].word[2]	VSR[VRT+32].word[3]
	0	32	64	96 127

Vector Divide Extended Signed Word VX-form

vdivesw VRT,VRA,VRB

4	VRT	VRA	VRB	907
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
   dividend ← EXTS(VSR[VRA+32].word[i]) << 32
   divisor  ← EXTS(VSR[VRB+32].word[i])

   VSR[VRT+32].word[i] ← CHOP32(dividend ÷ divisor)
end
```

For each integer value `i` from 0 to 3, do the following.

Let `dividend` be the signed integer value in word element `i` of `VSR[VRA+32]`, shifted left by 32 bits.

Let `divisor` be the signed integer value in word element `i` of `VSR[VRB+32]`.

`dividend` is divided by `divisor`.

The quotient is the unique signed integer that satisfies

$$dividend = (quotient \times divisor) + remainder$$

where $0 \leq remainder < |divisor|$ if the dividend is nonnegative, and $-|divisor| < remainder \leq 0$ if the dividend is negative.

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

$$\langle anything \rangle \div 0$$

the quotient is undefined.

The quotient is placed into word element `i` of `VSR[VRT+32]`.

Special Registers Altered:

None

Vector Divide Extended Unsigned Word VX-form

vdiveuw VRT,VRA,VRB

4	VRT	VRA	VRB	651
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
   dividend ← EXTZ(VSR[VRA+32].word[i]) << 32
   divisor  ← EXTZ(VSR[VRB+32].word[i])

   VSR[VRT+32].word[i] ← CHOP32(dividend ÷ divisor)
end
```

For each integer value `i` from 0 to 3, do the following.

Let `dividend` be the unsigned integer value in word element `i` of `VSR[VRA+32]`, shifted left by 32 bits.

Let `divisor` be the unsigned integer value in word element `i` of `VSR[VRB+32]`.

The quotient is the unique unsigned integer that satisfies

$$dividend = (quotient \times divisor) + remainder$$

where $0 \leq remainder < |divisor|$.

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

$$\langle anything \rangle \div 0$$

the quotient is undefined.

The quotient is placed into word element `i` of `VSR[VRT+32]`.

Special Registers Altered:

None

Register Data Layout for vdivesw & vdiveuw

src1	VSR[VRA+32].word[0]	VSR[VRA+32].word[1]	VSR[VRA+32].word[2]	VSR[VRA+32].word[3]
src2	VSR[VRB+32].word[0]	VSR[VRB+32].word[1]	VSR[VRB+32].word[2]	VSR[VRB+32].word[3]

result	VSR[VRT+32].word[0]	VSR[VRT+32].word[1]	VSR[VRT+32].word[2]	VSR[VRT+32].word[3]
	0	32	64	96 127

Vector Divide Signed Doubleword VX-form

vdivsd VRT,VRA,VRB

4	VRT	VRA	VRB	459
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 1
   dividend ← EXTS(VSR[VRA+32].dword[i])
   divisor  ← EXTS(VSR[VRB+32].dword[i])

   VSR[VRT+32].dword[i] ← CHOP64(dividend ÷ divisor)
end
```

For each integer value `i` from 0 to 1, do the following.

The signed integer in doubleword element `i` of `VSR[VRA+32]` is divided by the signed integer in doubleword element `i` of `VSR[VRB+32]`.

The quotient is the unique signed integer that satisfies

$$dividend = (quotient \times divisor) + remainder$$

where $0 \leq remainder < |divisor|$ if the dividend is nonnegative, and $-|divisor| < remainder \leq 0$ if the dividend is negative.

If an attempt is made to perform any of the divisions

0x8000_0000_0000_0000 ÷ -1
<anything> ÷ 0

then the quotient is undefined.

The quotient is placed into word element `i` of `VSR[VRT+32]`.

Special Registers Altered:

None

Vector Divide Unsigned Doubleword VX-form

vdivud VRT,VRA,VRB

4	VRT	VRA	VRB	203
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 1
   dividend ← EXTZ(VSR[VRA+32].dword[i])
   divisor  ← EXTZ(VSR[VRB+32].dword[i])

   VSR[VRT+32].dword[i] ← CHOP64(dividend ÷ divisor)
end
```

For each integer value `i` from 0 to 1, do the following.

The unsigned integer in doubleword element `i` of `VSR[VRA+32]` is divided by the unsigned integer in doubleword element `i` of `VSR[VRB+32]`.

The quotient is the unique unsigned integer that satisfies

$$dividend = (quotient \times divisor) + remainder$$

where $0 \leq remainder < divisor$.

If an attempt is made to perform the division

<anything> ÷ 0

then the quotient is undefined.

The quotient is placed into doubleword element `i` of `VSR[VRT+32]`.

Special Registers Altered:

None

Register Data Layout for vdivsd & vdivud

src1	VSR[VRA+32].dword[0]	VSR[VRA+32].dword[1]
src2	VSR[VRB+32].dword[0]	VSR[VRB+32].dword[1]
result	VSR[VRT+32].dword[0]	VSR[VRT+32].dword[1]
	0	64 127

Vector Divide Extended Signed Doubleword VX-form

vdivesd VRT,VRA,VRB

4	VRT	VRA	VRB	971
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 1
   dividend ← EXTS(VSR[VRA+32].dword[i]) << 64
   divisor  ← EXTS(VSR[VRB+32].dword[i])

   VSR[VRT+32].dword[i] ← CHOP64(dividend ÷ divisor)
end
```

For each integer value `i` from 0 to 1, do the following.

Let `dividend` be the signed integer value in doubleword element `i` of `VSR[VRA+32]`, shifted left by 64 bits.

Let `divisor` be the signed integer value in doubleword element `i` of `VSR[VRB+32]`.

`dividend` is divided by `divisor`.

The quotient is the unique signed integer that satisfies

$$dividend = (quotient \times divisor) + remainder$$

where $0 \leq remainder < |divisor|$ if the dividend is nonnegative, and $-|divisor| < remainder \leq 0$ if the dividend is negative.

If the quotient cannot be represented in 64 bits, or if an attempt is made to perform the division, $\langle anything \rangle \div 0$, the quotient is undefined.

The quotient is placed into doubleword element `i` of `VSR[VRT+32]`.

Special Registers Altered:

None

Vector Divide Extended Unsigned Doubleword VX-form

vdiveud VRT,VRA,VRB

4	VRT	VRA	VRB	715
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 1
   dividend ← EXTZ(VSR[VRA+32].dword[i]) << 64
   divisor  ← EXTZ(VSR[VRB+32].dword[i])

   VSR[VRT+32].dword[i] ← CHOP64(dividend ÷ divisor)
end
```

For each integer value `i` from 0 to 1, do the following.

Let `dividend` be the unsigned integer value in doubleword element `i` of `VSR[VRA+32]`, shifted left by 64 bits.

Let `divisor` be the unsigned integer value in doubleword element `i` of `VSR[VRB+32]`.

The quotient is the unique unsigned integer that satisfies

$$dividend = (quotient \times divisor) + remainder$$

where $0 \leq remainder < |divisor|$.

If the quotient cannot be represented in 64 bits, or if an attempt is made to perform the division,

$$\langle anything \rangle \div 0$$

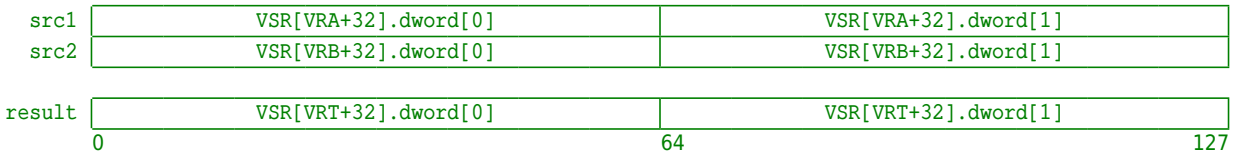
the quotient is undefined.

The quotient is placed into doubleword element `i` of `VSR[VRT+32]`.

Special Registers Altered:

None

Register Data Layout for vdivesd & vdiveud



Vector Divide Signed Quadword VX-form

vdivsq VRT,VRA,VRB

4	VRT	VRA	VRB	267
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

dividend    ← EXTS(VSR[VRA+32])
divisor     ← EXTS(VSR[VRB+32])

VSR[VRT+32] ← CHOP128(dividend ÷ divisor)
```

Let src1 be the signed integer value in VSR[VRA+32].

Let src2 be the signed integer value in VSR[VRB+32].

The quotient of src1 divided by src2 is placed into VSR[VRT+32].

The quotient is the unique signed integer that satisfies

$$dividend = (quotient \times divisor) + r$$

where $0 \leq remainder < |divisor|$ if the dividend is nonnegative, and $-|divisor| < remainder \leq 0$ if the dividend is negative.

If an attempt is made to perform any of the divisions

$$0x8000\_0000\_0000\_0000\_0000\_0000\_0000\_0000 \div -1$$
$$<anything> \div 0$$

then the contents of VSR[VRT+32] are undefined.

Special Registers Altered:

None

Vector Divide Unsigned Quadword VX-form

vdivuq VRT,VRA,VRB

4	VRT	VRA	VRB	11
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

dividend    ← EXTZ(VSR[VRA+32])
divisor     ← EXTZ(VSR[VRB+32])

VSR[VRT+32] ← CHOP128(dividend ÷ divisor)
```

Let src1 be the unsigned integer value in VSR[VRA+32].

Let src2 be the unsigned integer value in VSR[VRB+32].

The quotient of src1 divided by src2 is placed into VSR[VRT+32].

The quotient is the unique unsigned integer that satisfies

$$dividend = (quotient \times divisor) + r$$

where $0 \leq remainder < divisor$.

If an attempt is made to perform the division

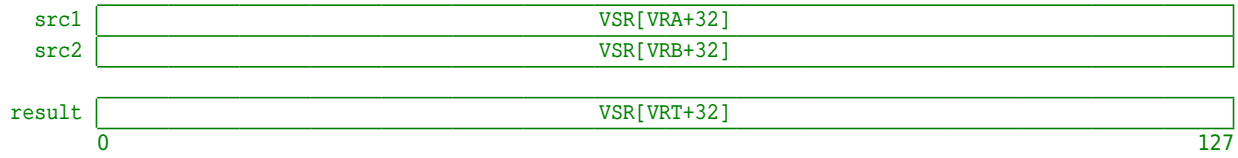
$$<anything> \div 0$$

then the contents of VSR[VRT+32] are undefined.

Special Registers Altered:

None

Register Data Layout for vdivsq & vdivuq



Vector Divide Extended Signed Quadword VX-form

vdivesq VRT,VRA,VRB

4	VRT	VRA	VRB	779
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

dividend    ← EXTS(VSR[VRA+32]) << 128
divisor     ← EXTS(VSR[VRB+32])

VSR[VRT+32] ← CHOP128(dividend ÷ divisor)
```

Let src1 be the signed integer value in VSR[VRA+32] concatenated with 128 0s.

Let src2 be the signed integer value in VSR[VRB+32].

The quotient of src1 divided by src2 is placed into VSR[VRT+32].

The quotient is the unique signed integer that satisfies

$$dividend = (quotient \times divisor) + r$$

where $0 \leq remainder < |divisor|$ if dividend is non-negative, and $-|divisor| < remainder \leq 0$ if dividend is negative.

If the quotient cannot be represented in 128 bits, or if an attempt is made to perform the division

$$\langle anything \rangle \div 0$$

then the contents of VSR[VRT+32] are undefined.

Special Registers Altered:

None

Vector Divide Extended Unsigned Quadword VX-form

vdiveuq VRT,VRA,VRB

4	VRT	VRA	VRB	523
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

dividend    ← EXTZ(VSR[VRA+32]) << 128
divisor     ← EXTZ(VSR[VRB+32])

VSR[VRT+32] ← CHOP128(dividend ÷ divisor)
```

Let src1 be the unsigned integer value in VSR[VRA+32] concatenated with 128 0s.

Let src2 be the unsigned integer value in VSR[VRB+32].

The quotient of src1 divided by src2 is placed into VSR[VRT+32].

The quotient is the unique unsigned integer that satisfies

$$dividend = (quotient \times divisor) + r$$

where $0 \leq remainder < divisor$.

If the quotient cannot be represented in 128 bits, or if an attempt is made to perform the division

$$\langle anything \rangle \div 0$$

then the contents of VSR[VRT+32] are undefined.

Special Registers Altered:

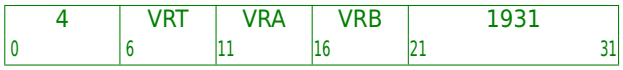
None

Register Data Layout for vdivesq & vdiveuq

6.9.1.6 Vector Integer Modulo Instructions

Vector Modulo Signed Word VX-form

vmodsw VRT,VRA,VRB




```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
   dividend ← EXTS(VSR[VRA+32].word[i])
   divisor  ← EXTS(VSR[VRB+32].word[i])

   VSR[VRT+32].word[i] ← CHOP32(dividend % divisor)
end
```

For each integer value `i` from 0 to 3, do the following.

The signed integer in word element `i` of `VSR[VRA+32]` is divided by the signed integer in word element `i` of `VSR[VRB+32]`.

The remainder is the unique signed integer that satisfies

$$\text{quotient} = \text{dividend} \div \text{divisor}$$
$$\text{remainder} = \text{dividend} - (\text{quotient} \times \text{divisor})$$

where $0 \leq \text{remainder} < |\text{divisor}|$ if the dividend is nonnegative, and $-|\text{divisor}| < \text{remainder} \leq 0$ if the dividend is negative.

If an attempt is made to perform any of the modulo operations

```
0x8000_0000 % -1
<anything> % 0
```

then the remainder is undefined.

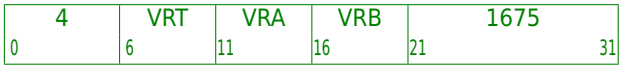
The remainder is placed into word element `i` of `VSR[VRT+32]`

Special Registers Altered:

None

Vector Modulo Unsigned Word VX-form

vmoduw VRT,VRA,VRB




```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
   dividend ← EXTZ(VSR[VRA+32].word[i])
   divisor  ← EXTZ(VSR[VRB+32].word[i])

   VSR[VRT+32].word[i] ← CHOP32(dividend % divisor)
end
```

For each integer value `i` from 0 to 3, do the following.

The unsigned integer in word element `i` of `VSR[VRA+32]` is divided by the unsigned integer in word element `i` of `VSR[VRB+32]`.

The remainder is the unique unsigned integer that satisfies

$$\text{quotient} = \text{dividend} \div \text{divisor}$$
$$\text{remainder} = \text{dividend} - (\text{quotient} \times \text{divisor})$$

where $0 \leq \text{remainder} < \text{divisor}$.

If an attempt is made to perform the modulo operation

```
<anything> % 0
```

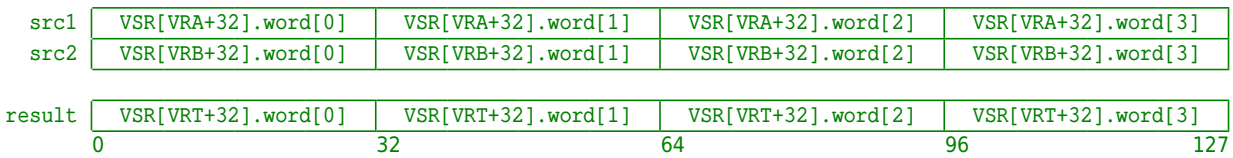
then the remainder is undefined.

The remainder is placed into word element `i` of `VSR[VRT+32]`.

Special Registers Altered:

None

Register Data Layout for vmodsw & vmoduw



Vector Modulo Signed Doubleword VX-form

vmodsd VRT,VRA,VRB

4	VRT	VRA	VRB	1995
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 1
   dividend ← EXTS(VSR[VRA+32].dword[i])
   divisor  ← EXTS(VSR[VRB+32].dword[i])

   VSR[VRT+32].dword[i] ← CHOP64(dividend % divisor)
end
```

For each integer value `i` from 0 to 1, do the following.

The signed integer in doubleword element `i` of `VSR[VRA+32]` is divided by the signed integer in doubleword element `i` of `VSR[VRB+32]`.

The remainder is the unique signed integer that satisfies

$$quotient = dividend \div divisor$$
$$remainder = dividend - (quotient \times divisor)$$

where $0 \leq remainder < |divisor|$ if the dividend is nonnegative, and $-|divisor| < remainder \leq 0$ if the dividend is negative.

If an attempt is made to perform any of the modulo operations

```
0x8000_0000_0000_0000 % -1
<anything> % 0
```

the remainder is undefined.

The remainder is placed into doubleword element `i` of `VSR[VRT+32]`.

Special Registers Altered:

None

Vector Modulo Unsigned Doubleword VX-form

vmodud VRT,VRA,VRB

4	VRT	VRA	VRB	1739
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 1
   dividend ← EXTZ(VSR[VRA+32].dword[i])
   divisor  ← EXTZ(VSR[VRB+32].dword[i])

   VSR[VRT+32].dword[i] ← CHOP64(dividend % divisor)
end
```

For each integer value `i` from 0 to 1, do the following.

The unsigned integer in doubleword element `i` of `VSR[VRA+32]` is divided by the unsigned integer in doubleword element `i` of `VSR[VRB+32]`.

The remainder is the unique unsigned integer that satisfies

$$quotient = dividend \div divisor$$
$$remainder = dividend - (quotient \times divisor)$$

where $0 \leq remainder < divisor$.

If an attempt is made to perform the modulo operation

```
<anything> % 0
```

the remainder is undefined.

The remainder is placed into doubleword element `i` of `VSR[VRT+32]`.

Special Registers Altered:

None

Register Data Layout for vmodsd & vmodud

src1	VSR[VRA+32].dword[0]	VSR[VRA+32].dword[1]
src2	VSR[VRB+32].dword[0]	VSR[VRB+32].dword[1]

result	VSR[VRT+32].dword[0]	VSR[VRT+32].dword[1]
	0 64	127

Vector Modulo Signed Quadword VX-form

vmodsq VRT,VRA,VRB

4	VRT	VRA	VRB	1803
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

dividend    ← EXTS(VSR[VRA+32])
divisor     ← EXTS(VSR[VRB+32])

VSR[VRT+32] ← CHOP128(dividend % divisor)
```

Let src1 be the signed integer value in VSR[VRA+32].
Let src2 be the signed integer value in VSR[VRB+32].

The remainder of src1 divided by src2 is placed into VSR[VRT+32].

The remainder is the unique signed integer that satisfies

$$quotient = dividend \div divisor$$
$$remainder = dividend - (quotient \times divisor)$$

where $0 \leq remainder < |divisor|$ if the dividend is nonnegative, and $-|divisor| < remainder \leq 0$ if the dividend is negative.

If an attempt is made to perform any of the modulo operations

```
0x8000_0000_0000_0000_0000_0000_0000_0000 % -1
<anything> % 0
```

then the contents of VSR[VRT+32] are undefined.

Special Registers Altered:

None

Vector Modulo Unsigned Quadword VX-form

vmoduq VRT,VRA,VRB

4	VRT	VRA	VRB	1547
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

dividend    ← EXTZ(VSR[VRA+32])
divisor     ← EXTZ(VSR[VRB+32])

VSR[VRT+32] ← CHOP128(dividend % divisor)
```

Let src1 be the unsigned integer value in VSR[VRA+32].
Let src2 be the unsigned integer value in VSR[VRB+32].

The remainder of src1 divided by src2 is placed into VSR[VRT+32].

The remainder is the unique unsigned integer that satisfies

$$quotient = dividend \div divisor$$
$$remainder = dividend - (quotient \times divisor)$$

where $0 \leq remainder < divisor$.

If an attempt is made to perform the modulo operation

```
<anything> % 0
```

then the contents of VSR[VRT+32] are undefined.

Special Registers Altered:

None

Register Data Layout for vmodsq & vmoduq

6.9.1.7 Vector Integer Sum-Across Instructions

Vector Sum across Signed Word Saturate VX-form

vsumsws VRT,VRA,VRB

4	VRT	VRA	VRB	1928
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

temp ← EXTS(VSR[VRB+32].word[3])
do i = 0 to 3
   temp ← temp + EXTS(VSR[VRA+32].word[i])
end
VSR[VRT+32].word[0] ← 0x0000_0000
VSR[VRT+32].word[1] ← 0x0000_0000
VSR[VRT+32].word[2] ← 0x0000_0000
VSR[VRT+32].word[3] ← si32_CLAMP(temp)
VSCR.SAT ← sat_flag
```

The sum of the signed integer values in the four word elements of VSR[VRA+32] is added to the signed integer value in the word element 3 of VSR[VRB+32].

- If the intermediate result is greater than $2^{31}-1$, the result saturates to $2^{31}-1$ and SAT is set to 1.
- If the intermediate result is less than -2^{31}, the result saturates to -2^{31} and SAT is set to 1.

The low-order 32 bits of the result are placed into word element 3 of VSR[VRT+32].

Word elements 0 to 2 of VSR[VRT+32] are set to 0.

Special Registers Altered:

Register	*Field(s)*
VSCR	SAT

Register Data Layout for vsumsws

src1	VSR[VRA+32].word[0]	VSR[VRA+32].word[1]	VSR[VRA+32].word[2]	VSR[VRA+32].word[3]
src2	unused	unused	unused	VSR[VRB+32].word[3]

result	0x0000_0000	0x0000_0000	0x0000_0000	VSR[VRT+32].word[3]
	0	32	64	96 127

Vector Sum across Half Signed Word Saturate VX-form

vsum2sws VRT,VRA,VRB

4	VRT	VRA	VRB	1672
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 1
   temp ← EXTS(VSR[VRB+32].dword[i].word[1])
   do j = 0 to 1
      temp ← temp +
         EXTS(VSR[VRA+32].dword[i].word[j])
   end
   VSR[VRT+32].dword[i].word[0] ← 0x0000_0000
   VSR[VRT+32].dword[i].word[1] ← si32_CLAMP(temp)
   VSCR.SAT ← sat_flag
end
```

Word elements 0 and 2 of VSR[VRT+32] are set to 0.

The sum of the signed integer values in word elements 0 and 1 in VSR[VRA+32] is added to the signed integer value in word element 1 of VSR[VRB+32].

- If the intermediate result is greater than $2^{31}-1$, the result saturates to $2^{31}-1$ and SAT is set to 1.
- If the intermediate result is less than -2^{31}, the result saturates to -2^{31} and SAT is set to 1.

The low-order 32 bits of the result are placed into word element 1 of VSR[VRT+32].

The sum of the signed integer values in word elements 2 and 3 in VSR[VRA+32] is added to the signed integer value in word element 3 of VSR[VRB+32].

- If the intermediate result is greater than $2^{31}-1$, the result saturates to $2^{31}-1$ and SAT is set to 1.
- If the intermediate result is less than -2^{31}, the result saturates to -2^{31} and SAT is set to 1.

The low-order 32 bits of the result are placed into word element 3 of VSR[VRT+32].

Special Registers Altered:

Register	*Field(s)*
VSCR	SAT

Register Data Layout for vsum2sws

src1	VSR[VRA+32].word[0]	VSR[VRA+32].word[1]	VSR[VRA+32].word[2]	VSR[VRA+32].word[3]
src2	unused	VSR[VRB+32].word[1]	unused	VSR[VRB+32].word[3]

result	0x0000_0000	VSR[VRT+32].word[1]	0x0000_0000	VSR[VRT+32].word[3]
	0	32	64	96 127

Vector Sum across Quarter Signed Byte Saturate VX-form

vsum4sbs VRT,VRA,VRB

4	VRT	VRA	VRB	1800
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
   temp ← EXTS(VSR[VRB+32].word[i])
   do j = 0 to 3
      temp ← temp + EXTS(VSR[VRA+32].word[i].byte[j])
   end
   VSR[VRT+32].word[i] ← si32_CLAMP(temp)
   VSCR.SAT ← sat_flag
end
```

For each integer value `i` from 0 to 3, do the following.

The sum of the signed integer values in the four byte elements contained in word element `i` of `VSR[VRA+32]` is added to the signed integer value in word element `i` of `VSR[VRB+32]`.

- If the intermediate result is greater than $2^{31}-1$, the result saturates to $2^{31}-1$ and `SAT` is set to 1.
- If the intermediate result is less than -2^{31}, the result saturates to -2^{31} and `SAT` is set to 1.

The result is placed into word element `i` of `VSR[VRT+32]`.

Special Registers Altered:

Register	*Field(s)*
VSCR	SAT

Vector Sum across Quarter Signed Halfword Saturate VX-form

vsum4shs VRT,VRA,VRB

4	VRT	VRA	VRB	1608
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
   temp ← EXTS(VSR[VRB+32].word[i])
   do j = 0 to 1
      temp ← temp + EXTS(VSR[VRA+32].word[i].hword[j])
   end
   VSR[VRT+32].word[i] ← si32_CLAMP(temp)
   VSCR.SAT ← sat_flag
end
```

For each integer value `i` from 0 to 3, do the following.

The sum of the signed integer values in the two halfword elements contained in word element `i` of `VSR[VRA+32]` is added to the signed integer value in word element `i` of `VSR[VRB+32]`.

- If the intermediate result is greater than $2^{31}-1$, the result saturates to $2^{31}-1$ and `SAT` is set to 1.
- If the intermediate result is less than -2^{31}, the result saturates to -2^{31} and `SAT` is set to 1.

The result is placed into the corresponding word element of `VSR[VRT+32]`.

Special Registers Altered:

Register	*Field(s)*
VSCR	SAT

Register Data Layout for vsum4sbs

src1	.byte[0]	.byte[1]	.byte[2]	.byte[3]	.byte[4]	.byte[5]	.byte[6]	.byte[7]	.byte[8]	.byte[9]	.byte[10]	.byte[11]	.byte[12]	.byte[13]	.byte[14]	.byte[15]
src2	VSR[VRB+32].word[0]				VSR[VRB+32].word[1]				VSR[VRB+32].word[2]				VSR[VRB+32].word[3]			
result	VSR[VRT+32].word[0]				VSR[VRT+32].word[1]				VSR[VRT+32].word[2]				VSR[VRT+32].word[3]			

0 8 16 24 32 40 48 56 64 72 80 88 96 104 112 120 127

Register Data Layout for vsum4shs

src1	VSR[VRA+32].hword[0]	VSR[VRA+32].hword[1]	VSR[VRA+32].hword[2]	VSR[VRA+32].hword[3]	VSR[VRA+32].hword[4]	VSR[VRA+32].hword[5]	VSR[VRA+32].hword[6]	VSR[VRA+32].hword[7]
src2	VSR[VRB+32].word[0]		VSR[VRB+32].word[1]		VSR[VRB+32].word[2]		VSR[VRB+32].word[3]	
result	VSR[VRT+32].word[0]		VSR[VRT+32].word[1]		VSR[VRT+32].word[2]		VSR[VRT+32].word[3]	

0 16 32 48 64 80 96 112 127

Vector Sum across Quarter Unsigned Byte Saturate VX-form

vsum4ubs VRT,VRA,VRB

4	VRT	VRA	VRB	1544
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
   temp ← EXTZ(VSR[VRB+32].word[i])
   do j = 0 to 3
      temp ← temp + EXTZ(VSR[VRA+32].word[i].byte[j])
   end
   VSR[VRT+32].word[i] ← ui32_CLAMP(temp)
   VSCR.SAT ← sat_flag
end
```

For each integer value `i` from 0 to 3, do the following.

The sum of the unsigned integer values in the four byte elements contained in word element `i` of `VSR[VRA+32]` is added to the unsigned integer value in word element `i` of `VSR[VRB+32]`.

- If the intermediate result is greater than $2^{32}-1$, it saturates to $2^{32}-1$ and `SAT` is set to 1.

The result is placed into word element `i` of `VSR[VRT+32]`.

Special Registers Altered:

Register	*Field(s)*
VSCR	SAT

Register Data Layout for vsum4ubs

src1	.byte[0]	.byte[1]	.byte[2]	.byte[3]	.byte[4]	.byte[5]	.byte[6]	.byte[7]	.byte[8]	.byte[9]	.byte[10]	.byte[11]	.byte[12]	.byte[13]	.byte[14]	.byte[15]
src2	VSR[VRB+32].word[0]				VSR[VRB+32].word[1]				VSR[VRB+32].word[2]				VSR[VRB+32].word[3]			

result	VSR[VRT+32].word[0]				VSR[VRT+32].word[1]				VSR[VRT+32].word[2]				VSR[VRT+32].word[3]			
	0	8	16	24	32	40	48	56	64	72	80	88	96	104	112	120 127

6.9.1.8 Vector Integer Negate Instructions

Vector Negate Word VX-form

vnegw VRT,VRB

4	VRT	6	VRB	1538
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
   src ← EXTS(VSR[VRB+32].word[i])

   VSR[VRT+32].word[i] ← CHOP32(¬src + 1)
end
```

For each integer value `i` from 0 to 3, do the following.

The sum of the one's-complement of the signed integer in word element `i` of `VSR[VRB+32]` and 1 is placed into word element `i` of `VSR[VRT+32]`.

Special Registers Altered:

None

Vector Negate Doubleword VX-form

vnegd VRT,VRB

4	VRT	7	VRB	1538
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 1
   src ← EXTS(VSR[VRB+32].dword[i])

   VSR[VRT+32]dword[i] ← CHOP64(¬src + 1)
end
```

For each integer value `i` from 0 to 1, do the following.

The sum of the one's-complement of the signed integer in doubleword element `i` of `VSR[VRB+32]` and 1 is placed into doubleword element `i` of `VSR[VRT+32]`.

Special Registers Altered:

None

Register Data Layout for vnegw

src	VSR[VRB+32].word[0]	VSR[VRB+32].word[1]	VSR[VRB+32].word[2]	VSR[VRB+32].word[3]
result	VSR[VRT+32].word[0]	VSR[VRT+32].word[1]	VSR[VRT+32].word[2]	VSR[VRT+32].word[3]
	0	32	64	96 127

Register Data Layout for vnegd

src	VSR[VRB+32].dword[0]	VSR[VRB+32].dword[1]
result	VSR[VRT+32].dword[0]	VSR[VRT+32].dword[1]
	0	64 127

6.9.1.9 Vector Extend Sign Instructions

Vector Extend Sign Byte To Word VX-form

vextsb2w VRT,VRB

4	VRT	16	VRB	1538
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
   src ← VSR[VRB+32].word[i].bit[24:31]

   VSR[VRT+32].word[i] ← EXTS32(src)
end
```

For each integer value `i` from 0 to 3, do the following.

The signed integer in bits 24:31 of word element `i` of `VSR[VRB+32]` is sign-extended and placed into word element `i` of `VSR[VRT+32]`.

Special Registers Altered:

None

Vector Extend Sign Halfword To Word VX-form

vextsh2w VRT,VRB

4	VRT	17	VRB	1538
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
   src ← VSR[VRB+32].word[i].bit[16:31]

   VSR[VRT+32].word[i] ← EXTS32(src)
end
```

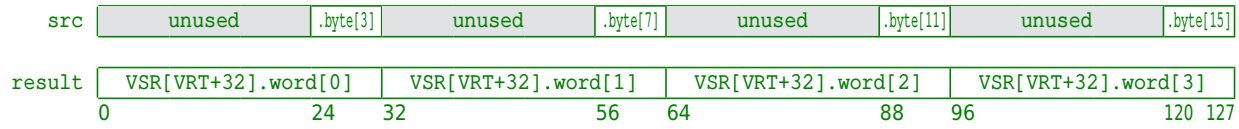
For each integer value `i` from 0 to 3, do the following.

The signed integer in bits 16:31 of word element `i` of `VSR[VRB+32]` is sign-extended and placed into word element `i` of `VSR[VRT+32]`.

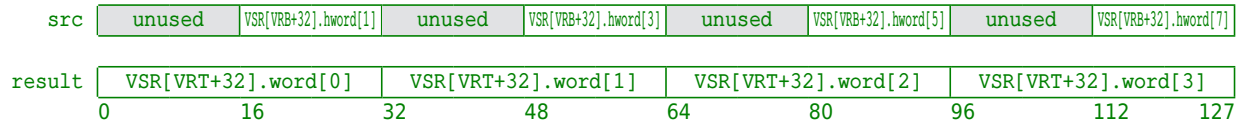
Special Registers Altered:

None

Register Data Layout for vextsb2w



Register Data Layout for vextsh2w



Vector Extend Sign Byte To Doubleword VX-form

vextsb2d VRT,VRB

4	VRT	24	VRB	1538
0 6	11	16	21	31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 1
   src ← VSR[VRB+32].dword[i].bit[56:63]

   VSR[VRT+32].dword[i] ← EXTS64(src)
end
```

For each integer value `i` from 0 to 1, do the following.

The signed integer in bits 56:63 of doubleword element `i` of `VSR[VRB+32]` is sign-extended and placed into doubleword element `i` of `VSR[VRT+32]`.

Special Registers Altered:

None

Vector Extend Sign Halfword To Doubleword VX-form

vextsh2d VRT,VRB

4	VRT	25	VRB	1538
0 6	11	16	21	31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 1
   src ← VSR[VRB+32].dword[i].bit[48:63]

   VSR[VRT+32].dword[i] ← EXTS64(src)
end
```

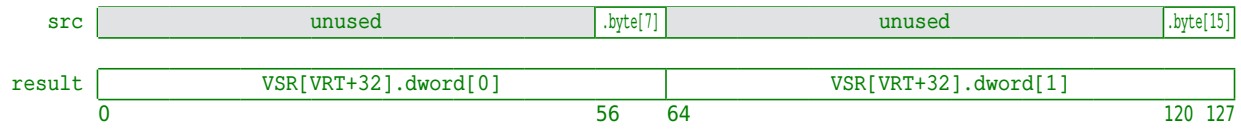
For each integer value `i` from 0 to 1, do the following.

The signed integer in bits 48:63 of doubleword element `i` of `VSR[VRB+32]` is sign-extended and placed into doubleword element `i` of `VSR[VRT+32]`.

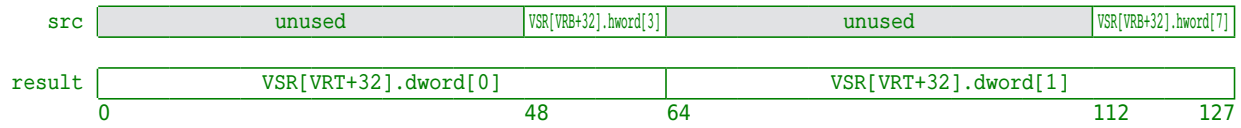
Special Registers Altered:

None

Register Data Layout for vextsb2d



Register Data Layout for vextsh2d



Vector Extend Sign Word To Doubleword VX-form

vextsw2d VRT,VRB

4	VRT	26	VRB	1538
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 1
   src ← VSR[VRB+32].dword[i].bit[32:63]

   VSR[VRT+32].dword[i] ← EXTS64(src)
end
```

For each integer value `i` from 0 to 1, do the following.

The signed integer in bits 32:63 of doubleword element `i` of `VSR[VRB+32]` is sign-extended and placed into doubleword element `i` of `VSR[VRT+32]`.

Special Registers Altered:

None

Vector Extend Sign Doubleword to Quadword VX-form

vextsd2q VRT,VRB

4	VRT	27	VRB	1538
0	6	11	16	21 31

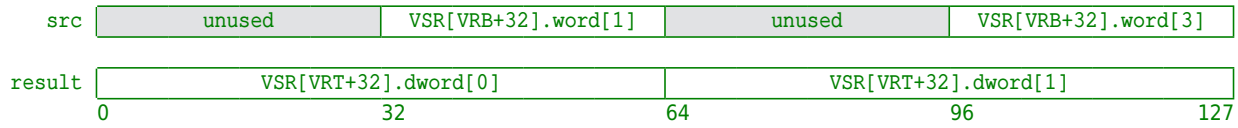
```
if MSR.VEC=0 then Vector_Unavailable()

VSR[VRT+32] ← EXTS128(VSR[VRB+32].bit[64:127])
```

The signed integer in bits 64:127 of `VSR[VRB+32]` is signed extended to 128 bits and placed into `VSR[VRT+32]`.

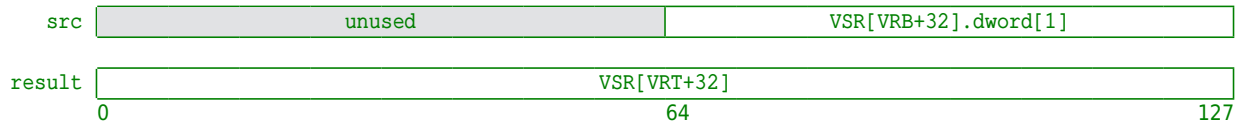
Special Registers Altered:

None

Register Data Layout for vextsw2d



Register Data Layout for vextsd2q



6.9.1.10 Vector Integer Average Instructions

Vector Average Signed Byte VX-form

vavgsb VRT,VRA,VRB

4	VRT	VRA	VRB	1282
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 15
   src1 ← EXTS(VSR[VRA+32].byte[i])
   src2 ← EXTS(VSR[VRB+32].byte[i])

   VSR[VRT+32].byte[i] ← CHOP8((src1 + src2 + 1) >> 1)
end
```

For each integer value `i` from 0 to 15, do the following.

Let `src1` be the signed integer value in byte element `i` of `VSR[VRA+32]`.

Let `src2` be the signed integer value in byte element `i` of `VSR[VRB+32]`.

`src1` is added to `src2`.

The sum is incremented by 1 and then shifted right 1 bit.

The low-order 8 bits of the result are placed into byte element `i` of `VSR[VRT+32]`.

Special Registers Altered:

None

Vector Average Unsigned Byte VX-form

vavgub VRT,VRA,VRB

4	VRT	VRA	VRB	1026
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 15
   src1 ← EXTZ(VSR[VRA+32].byte[i])
   src2 ← EXTZ(VSR[VRB+32].byte[i])

   VSR[VRT+32].byte[i] ← CHOP8((src1 + src2 + 1) >> 1)
end
```

For each integer value i from 0 to 15, do the following.

Let `src1` be the unsigned integer value in byte element `i` of `VSR[VRA+32]`.

Let `src2` be the unsigned integer value in byte element `i` of `VSR[VRB+32]`.

`src1` is added to `src2`.

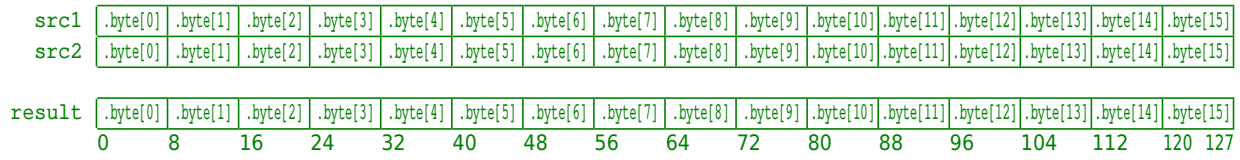
The sum is incremented by 1 and then shifted right 1 bit.

The low-order 8 bits of the result are placed into byte element `i` of `VSR[VRT+32]`.

Special Registers Altered:

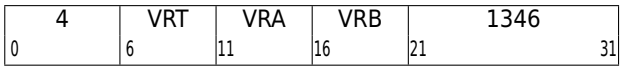
None

Register Data Layout for vavgsb & vavgub



Vector Average Signed Halfword VX-form

vavgsh VRT,VRA,VRB




```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 7
   src1 ← EXTS(VSR[VRA+32].hword[i])
   src2 ← EXTS(VSR[VRB+32].hword[i])

   VSR[VRT+32].hword[i] ←
      CHOP16((src1 + src2 + 1) >> 1)
end
```

For each integer value `i` from 0 to 7, do the following.

Let `src1` be the signed integer value in halfword element `i` of `VSR[VRA+32]`.

Let `src2` be the signed integer value in halfword element `i` of `VSR[VRB+32]`.

`src1` is added to `src2`.

The sum is incremented by 1 and then shifted right 1 bit.

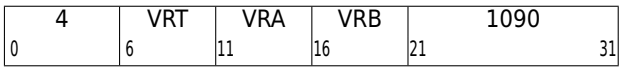
The low-order 16 bits of the result are placed into halfword element `i` of `VSR[VRT+32]`.

Special Registers Altered:

None

Vector Average Unsigned Halfword VX-form

vavguh VRT,VRA,VRB




```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 7
   src1 ← EXTZ(VSR[VRA+32].hword[i])
   src2 ← EXTZ(VSR[VRB+32].hword[i])

   VSR[VRT+32].hword[i] ←
      CHOP16((src1 + src2 + 1) >> 1)
end
```

For each integer value i from 0 to 7, do the following.

Let `src1` be the unsigned integer value in halfword element `i` of `VSR[VRA+32]`.

Let `src2` be the unsigned integer value in halfword element `i` of `VSR[VRB+32]`.

`src1` is added to `src2`.

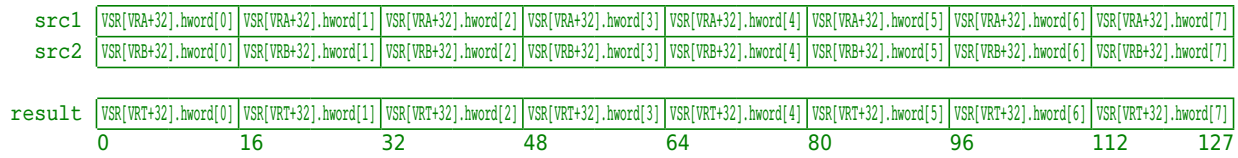
The sum is incremented by 1 and then shifted right 1 bit.

The low-order 16 bits of the result are placed into halfword element `i` of `VSR[VRT+32]`.

Special Registers Altered:

None

Register Data Layout for vavgsh & vavguh



Vector Average Signed Word VX-form

vavgsw VRT,VRA,VRB

4	VRT	VRA	VRB	1410
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
   src1 ← EXTS(VSR[VRA+32].word[i])
   src2 ← EXTS(VSR[VRB+32].word[i])

   VSR[VRT+32].word[i] ←
      Chop32((src1 + src2 + 1) >> 1)
end
```

For each integer value `i` from 0 to 3, do the following.

Let `src1` be the signed integer value in word element `i` of `VSR[VRA+32]`.

Let `src2` be the signed integer value in word element `i` of `VSR[VRB+32]`.

`src1` is added to `src2`.

The sum is incremented by 1 and then shifted right 1 bit.

The low-order 32 bits of the result are placed into word element `i` of `VSR[VRT+32]`.

Special Registers Altered:

None

Vector Average Unsigned Word VX-form

vavguw VRT,VRA,VRB

4	VRT	VRA	VRB	1154
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
   src1 ← EXTZ(VSR[VRA+32].word[i])
   src2 ← EXTZ(VSR[VRB+32].word[i])

   VSR[VRT+32].word[i] ←
      Chop32((src1 + src2 + 1) >> 1)
end
```

For each integer value i from 0 to 3, do the following.

Let `src1` be the unsigned integer value in word element `i` of `VSR[VRA+32]`.

Let `src2` be the unsigned integer value in word element `i` of `VSR[VRB+32]`.

`src1` is added to `src2`.

The sum is incremented by 1 and then shifted right 1 bit.

The low-order 32 bits of the result are placed into word element `i` of `VSR[VRT+32]`.

Special Registers Altered:

None

Register Data Layout for vavgsw & vavguw

src1	VSR[VRA+32].word[0]	VSR[VRA+32].word[1]	VSR[VRA+32].word[2]	VSR[VRA+32].word[3]
src2	VSR[VRB+32].word[0]	VSR[VRB+32].word[1]	VSR[VRB+32].word[2]	VSR[VRB+32].word[3]
result	VSR[VRT+32].word[0]	VSR[VRT+32].word[1]	VSR[VRT+32].word[2]	VSR[VRT+32].word[3]
	0	32	64	96 127

6.9.1.11 Vector Integer Absolute Difference Instructions

This section describes a set of instructions that return the absolute value of the difference of integer values.

Vector Absolute Difference Unsigned Byte VX-form

vabsdub VRT,VRA,VRB

4	VRT	VRA	VRB	1027
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 15
   src1 ← EXTZ(VSR[VRA+32].byte[i])
   src2 ← EXTZ(VSR[VRB+32].byte[i])

   if src1 > src2 then
      VSR[VRT+32].byte[i] ← CHOP8(src1 + ¬src2 + 1)
   else
      VSR[VRT+32].byte[i] ← CHOP8(src2 + ¬src1 + 1)
end
```

For each integer value `i` from 0 to 15, do the following.

Let `src1` be the unsigned integer value in byte element `i` of `VSR[VRA+32]`.

Let `src2` be the unsigned integer value in byte element `i` of `VSR[VRB+32]`.

`src1` is subtracted by `src2`.

The absolute value of the difference is placed into byte element `i` of `VSR[VRT+32]`.

Special Registers Altered:

None

Vector Absolute Difference Unsigned Halfword VX-form

vabsduh VRT,VRA,VRB

4	VRT	VRA	VRB	1091
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 7
   src1 ← EXTZ(VSR[VRA+32].hword[i])
   src2 ← EXTZ(VSR[VRB+32].hword[i])

   if src1 > src2 then
      VSR[VRT+32].hword[i] ← CHOP16(src1 + ¬src2 + 1)
   else
      VSR[VRT+32].hword[i] ← CHOP16(src2 + ¬src1 + 1)
end
```

For each integer value `i` from 0 to 7, do the following.

Let `src1` be the unsigned integer value in halfword element `i` of `VSR[VRA+32]`.

Let `src2` be the unsigned integer value in halfword element `i` of `VSR[VRB+32]`.

`src1` is subtracted by `src2`.

The absolute value of the difference is placed into halfword element `i` of `VSR[VRT+32]`.

Special Registers Altered:

None

Register Data Layout for vabsdub

src1	.byte[0]	.byte[1]	.byte[2]	.byte[3]	.byte[4]	.byte[5]	.byte[6]	.byte[7]	.byte[8]	.byte[9]	.byte[10]	.byte[11]	.byte[12]	.byte[13]	.byte[14]	.byte[15]
src2	.byte[0]	.byte[1]	.byte[2]	.byte[3]	.byte[4]	.byte[5]	.byte[6]	.byte[7]	.byte[8]	.byte[9]	.byte[10]	.byte[11]	.byte[12]	.byte[13]	.byte[14]	.byte[15]
result	.byte[0]	.byte[1]	.byte[2]	.byte[3]	.byte[4]	.byte[5]	.byte[6]	.byte[7]	.byte[8]	.byte[9]	.byte[10]	.byte[11]	.byte[12]	.byte[13]	.byte[14]	.byte[15]
	0	8	16	24	32	40	48	56	64	72	80	88	96	104	112	120 127

Register Data Layout for vabsduh

src1	VSR[VRA+32].hword[0]	VSR[VRA+32].hword[1]	VSR[VRA+32].hword[2]	VSR[VRA+32].hword[3]	VSR[VRA+32].hword[4]	VSR[VRA+32].hword[5]	VSR[VRA+32].hword[6]	VSR[VRA+32].hword[7]
src2	VSR[VRB+32].hword[0]	VSR[VRB+32].hword[1]	VSR[VRB+32].hword[2]	VSR[VRB+32].hword[3]	VSR[VRB+32].hword[4]	VSR[VRB+32].hword[5]	VSR[VRB+32].hword[6]	VSR[VRB+32].hword[7]
result	VSR[VRT+32].hword[0]	VSR[VRT+32].hword[1]	VSR[VRT+32].hword[2]	VSR[VRT+32].hword[3]	VSR[VRT+32].hword[4]	VSR[VRT+32].hword[5]	VSR[VRT+32].hword[6]	VSR[VRT+32].hword[7]
	0	16	32	48	64	80	96	112 127

Vector Absolute Difference Unsigned Word VX-form

vabsduw VRT,VRA,VRB

4	VRT	VRA	VRB	1155
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
   src1 ← EXTZ(VSR[VRA+32].word[i])
   src2 ← EXTZ(VSR[VRB+32].word[i])

   if src1 > src2 then
      VSR[VRT+32].word[i] ← CHOP32(src1 + ¬src2 + 1)
   else
      VSR[VRT+32].word[i] ← CHOP32(src2 + ¬src1 + 1)
end
```

For each integer value `i` from 0 to 3, do the following.

Let `src1` be the unsigned integer value in word element `i` of `VSR[VRA+32]`.

Let `src2` be the unsigned integer value in word element `i` of `VSR[VRB+32]`.

`src1` is subtracted by `src2`.

The absolute value of the difference is placed into word element `i` of `VSR[VRT+32]`.

Special Registers Altered:

None

Register Data Layout for vabsduw

src1	VSR[VRA+32].word[0]	VSR[VRA+32].word[1]	VSR[VRA+32].word[2]	VSR[VRA+32].word[3]
src2	VSR[VRB+32].word[0]	VSR[VRB+32].word[1]	VSR[VRB+32].word[2]	VSR[VRB+32].word[3]
result	VSR[VRT+32].word[0]	VSR[VRT+32].word[1]	VSR[VRT+32].word[2]	VSR[VRT+32].word[3]
	0	32	64	96 127

6.9.2 Vector Integer Maximum/Minimum Instructions

6.9.2.1 Vector Integer Maximum Instructions

Vector Maximum Signed Byte VX-form

vmaxsb VRT,VRA,VRB

4	VRT	VRA	VRB	258
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 15
   src1 ← VSR[VRA+32].byte[i]
   src2 ← VSR[VRB+32].byte[i]

   gt_flag ← EXTS(src1) > EXTS(src2)
   VSR[VRT+32].byte[i] ← gt_flag=1 ? src1 : src2
end
```

For each integer value i from 0 to 15, do the following.

Let src1 be the signed integer value in byte element i of VSR[VRA+32].

Let src2 be the signed integer value in byte element i of VSR[VRB+32].

src1 is compared to src2. The larger of the two values is placed into byte element i of VSR[VRT+32].

Special Registers Altered:

None

Vector Maximum Unsigned Byte VX-form

vmaxub VRT,VRA,VRB

4	VRT	VRA	VRB	2
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 15
   src1 ← VSR[VRA+32].byte[i]
   src2 ← VSR[VRB+32].byte[i]

   gt_flag ← EXTZ(src1) > EXTZ(src2)
   VSR[VRT+32].byte[i] ← gt_flag=1 ? src1 : src2
end
```

For each integer value i from 0 to 15, do the following.

Let src1 be the signed integer value in byte element i of VSR[VRA+32].

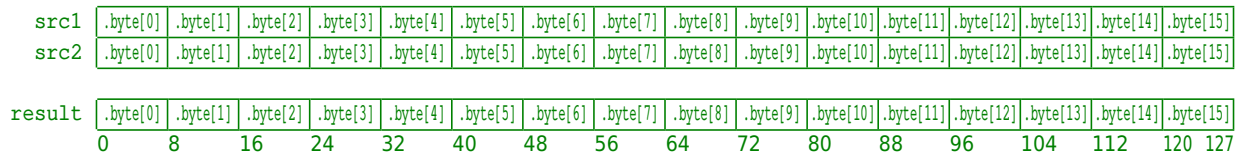
Let src2 be the signed integer value in byte element i of VSR[VRB+32].

src1 is compared to src2. The larger of the two values is placed into byte element i of VSR[VRT+32].

Special Registers Altered:

None

Register Data Layout for vmaxsb & vmaxub



Vector Maximum Signed Halfword VX-form

vmaxsh VRT,VRA,VRB

4	VRT	VRA	VRB	322
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 7
   src1 ← VSR[VRA+32].hword[i]
   src2 ← VSR[VRB+32].hword[i]

   gt_flag ← EXTS(src1) > EXTS(src2)
   VSR[VRT+32].hword[i] ← gt_flag=1 ? src1 : src2
end
```

For each integer value i from 0 to 7, do the following.

Let src1 be the signed integer value in halfword element i of VSR[VRA+32].

Let src2 be the signed integer value in halfword element i of VSR[VRB+32].

src1 is compared to src2. The larger of the two values is placed into halfword element i of VSR[VRT+32].

Special Registers Altered:

None

Vector Maximum Unsigned Halfword VX-form

vmaxuh VRT,VRA,VRB

4	VRT	VRA	VRB	66
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 7
   src1 ← VSR[VRA+32].hword[i]
   src2 ← VSR[VRB+32].hword[i]

   gt_flag ← EXTZ(src1) > EXTZ(src2)
   VSR[VRT+32].hword[i] ← gt_flag=1 ? src1 : src2
end
```

For each integer value i from 0 to 7, do the following.

Let src1 be the unsigned integer value in halfword element i of VSR[VRA+32].

Let src2 be the unsigned integer value in halfword element i of VSR[VRB+32].

src1 is compared to src2. The larger of the two values is placed into halfword element i of VSR[VRT+32].

Special Registers Altered:

None

Register Data Layout for vmaxsh & vmaxuh

src1	VSR[VRA+32].hword[0]	VSR[VRA+32].hword[1]	VSR[VRA+32].hword[2]	VSR[VRA+32].hword[3]	VSR[VRA+32].hword[4]	VSR[VRA+32].hword[5]	VSR[VRA+32].hword[6]	VSR[VRA+32].hword[7]
src2	VSR[VRB+32].hword[0]	VSR[VRB+32].hword[1]	VSR[VRB+32].hword[2]	VSR[VRB+32].hword[3]	VSR[VRB+32].hword[4]	VSR[VRB+32].hword[5]	VSR[VRB+32].hword[6]	VSR[VRB+32].hword[7]

result	VSR[VRT+32].hword[0]	VSR[VRT+32].hword[1]	VSR[VRT+32].hword[2]	VSR[VRT+32].hword[3]	VSR[VRT+32].hword[4]	VSR[VRT+32].hword[5]	VSR[VRT+32].hword[6]	VSR[VRT+32].hword[7]
	0	16	32	48	64	80	96	112 127

Vector Maximum Signed Word VX-form

vmaxsw VRT,VRA,VRB

4	VRT	VRA	VRB	386
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
   src1 ← VSR[VRA+32].word[i]
   src2 ← VSR[VRB+32].word[i]

   gt_flag ← EXTS(src1) > EXTS(src2)
   VSR[VRT+32].word[i] ← gt_flag=1 ? src1 : src2
end
```

For each integer value `i` from 0 to 3, do the following.

Let `src1` be the signed integer value in word element `i` of `VSR[VRA+32]`.

Let `src2` be the signed integer value in word element `i` of `VSR[VRB+32]`.

`src1` is compared to `src2`. The larger of the two values is placed into word element `i` of `VSR[VRT+32]`.

Special Registers Altered:

None

Vector Maximum Unsigned Word VX-form

vmaxuw VRT,VRA,VRB

4	VRT	VRA	VRB	130
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
   src1 ← VSR[VRA+32].word[i]
   src2 ← VSR[VRB+32].word[i]

   gt_flag ← EXTZ(src1) > EXTZ(src2)
   VSR[VRT+32].word[i] ← gt_flag=1 ? src1 : src2
end
```

For each integer value `i` from 0 to 3, do the following.

Let `src1` be the unsigned integer value in word element `i` of `VSR[VRA+32]`.

Let `src2` be the unsigned integer value in word element `i` of `VSR[VRB+32]`.

`src1` is compared to `src2`. The larger of the two values is placed into word element `i` of `VSR[VRT+32]`.

Special Registers Altered:

None

Register Data Layout for vmaxsw & vmaxuw

src1	VSR[VRA+32].word[0]	VSR[VRA+32].word[1]	VSR[VRA+32].word[2]	VSR[VRA+32].word[3]
src2	VSR[VRB+32].word[0]	VSR[VRB+32].word[1]	VSR[VRB+32].word[2]	VSR[VRB+32].word[3]

result	VSR[VRT+32].word[0]	VSR[VRT+32].word[1]	VSR[VRT+32].word[2]	VSR[VRT+32].word[3]
	0	32	64	96 127

Vector Maximum Signed Doubleword VX-form

vmaxsd VRT,VRA,VRB

4	VRT	VRA	VRB	450
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 1
   src1 ← VSR[VRA+32].dword[i]
   src2 ← VSR[VRB+32].dword[i]

   gt_flag ← EXTS(src1) > EXTS(src2)
   VSR[VRT+32].dword[i] ← gt_flag=1 ? src1 : src2
end
```

For each integer value `i` from 0 to 1, do the following.

Let `src1` be the signed integer value in doubleword element `i` of `VSR[VRA+32]`.

Let `src2` be the signed integer value in doubleword element `i` of `VSR[VRB+32]`.

`src1` is compared to `src2`. The larger of the two values is placed into doubleword element `i` of `VSR[VRT+32]`.

Special Registers Altered:

None

Vector Maximum Unsigned Doubleword VX-form

vmaxud VRT,VRA,VRB

4	VRT	VRA	VRB	194
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 1
   src1 ← VSR[VRA+32].dword[i]
   src2 ← VSR[VRB+32].dword[i]

   gt_flag ← EXTZ(src1) > EXTZ(src2)
   VSR[VRT+32].dword[i] ← gt_flag=1 ? src1 : src2
end
```

For each integer value `i` from 0 to 1, do the following.

Let `src1` be the unsigned integer value in doubleword element `i` of `VSR[VRA+32]`.

Let `src2` be the unsigned integer value in doubleword element `i` of `VSR[VRB+32]`.

`src1` is compared to `src2`. The larger of the two values is placed into doubleword element `i` of `VSR[VRT+32]`.

Special Registers Altered:

None

Register Data Layout for vmaxsd & vmaxud

src1	VSR[VRA+32].dword[0]	VSR[VRA+32].dword[1]
src2	VSR[VRB+32].dword[0]	VSR[VRB+32].dword[1]
result	VSR[VRT+32].dword[0]	VSR[VRT+32].dword[1]
	0 64	127

6.9.2.2 Vector Integer Minimum Instructions

Vector Minimum Signed Byte VX-form

vminsb VRT,VRA,VRB

4	VRT	VRA	VRB	770
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 15
   src1 ← VSR[VRA+32].byte[i]
   src2 ← VSR[VRB+32].byte[i]

   lt_flag ← EXTS(src1) < EXTS(src2)
   VSR[VRT+32].byte[i] ← lt_flag=1 ? src1 : src2
end
```

For each integer value i from 0 to 15, do the following.

Let src1 be the signed integer value in byte element i of VSR[VRA+32].

Let src2 be the signed integer value in byte element i of VSR[VRB+32].

src1 is compared to src2. The smaller of the two values is placed into byte element i of VSR[VRT+32].

Special Registers Altered:

None

Vector Minimum Unsigned Byte VX-form

vminub VRT,VRA,VRB

4	VRT	VRA	VRB	514
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 15
   src1 ← VSR[VRA+32].byte[i]
   src2 ← VSR[VRB+32].byte[i]

   lt_flag ← EXTZ(src1) < EXTZ(src2)
   VSR[VRT+32].byte[i] ← lt_flag=1 ? src1 : src2
end
```

For each integer value i from 0 to 15, do the following.

Let src1 be the unsigned integer value in byte element i of VSR[VRA+32].

Let src2 be the unsigned integer value in byte element i of VSR[VRB+32].

src1 is compared to src2. The smaller of the two values is placed into byte element i of VSR[VRT+32].

Special Registers Altered:

None

Register Data Layout for vminsb & vminub

src1	.byte[0]	.byte[1]	.byte[2]	.byte[3]	.byte[4]	.byte[5]	.byte[6]	.byte[7]	.byte[8]	.byte[9]	.byte[10]	.byte[11]	.byte[12]	.byte[13]	.byte[14]	.byte[15]
src2	.byte[0]	.byte[1]	.byte[2]	.byte[3]	.byte[4]	.byte[5]	.byte[6]	.byte[7]	.byte[8]	.byte[9]	.byte[10]	.byte[11]	.byte[12]	.byte[13]	.byte[14]	.byte[15]
result	.byte[0]	.byte[1]	.byte[2]	.byte[3]	.byte[4]	.byte[5]	.byte[6]	.byte[7]	.byte[8]	.byte[9]	.byte[10]	.byte[11]	.byte[12]	.byte[13]	.byte[14]	.byte[15]
	0	8	16	24	32	40	48	56	64	72	80	88	96	104	112	120 127

Vector Minimum Signed Halfword VX-form

vminsh VRT,VRA,VRB

4	VRT	VRA	VRB	834
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 7
   src1 ← VSR[VRA+32].hword[i]
   src2 ← VSR[VRB+32].hword[i]

   lt_flag ← EXTS(src1) < EXTS(src2)
   VSR[VRT+32].hword[i] ← lt_flag=1 ? src1 : src2
end
```

For each integer value i from 0 to 7, do the following.

Let src1 be the signed integer value in halfword element i of VSR[VRA+32].

Let src2 be the signed integer value in halfword element i of VSR[VRB+32].

src1 is compared to src2. The smaller of the two values is placed into halfword element i of VSR[VRT+32].

Special Registers Altered:

None

Vector Minimum Unsigned Halfword VX-form

vminuh VRT,VRA,VRB

4	VRT	VRA	VRB	578
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 7
   src1 ← VSR[VRA+32].hword[i]
   src2 ← VSR[VRB+32].hword[i]

   lt_flag ← EXTZ(src1) < EXTZ(src2)
   VSR[VRT+32].hword[i] ← lt_flag=1 ? src1 : src2
end
```

For each integer value i from 0 to 7, do the following.

Let src1 be the unsigned integer value in halfword element i of VSR[VRA+32].

Let src2 be the unsigned integer value in halfword element i of VSR[VRB+32].

src1 is compared to src2. The smaller of the two values is placed into halfword element i of VSR[VRT+32].

Special Registers Altered:

None

Register Data Layout for vminsh & vminuh

src1	VSR[VRA+32].hword[0]	VSR[VRA+32].hword[1]	VSR[VRA+32].hword[2]	VSR[VRA+32].hword[3]	VSR[VRA+32].hword[4]	VSR[VRA+32].hword[5]	VSR[VRA+32].hword[6]	VSR[VRA+32].hword[7]
src2	VSR[VRB+32].hword[0]	VSR[VRB+32].hword[1]	VSR[VRB+32].hword[2]	VSR[VRB+32].hword[3]	VSR[VRB+32].hword[4]	VSR[VRB+32].hword[5]	VSR[VRB+32].hword[6]	VSR[VRB+32].hword[7]

result	VSR[VRT+32].hword[0]	VSR[VRT+32].hword[1]	VSR[VRT+32].hword[2]	VSR[VRT+32].hword[3]	VSR[VRT+32].hword[4]	VSR[VRT+32].hword[5]	VSR[VRT+32].hword[6]	VSR[VRT+32].hword[7]
	0	16	32	48	64	80	96	112 127

Vector Minimum Signed Word VX-form

vminsw VRT,VRA,VRB

4	VRT	VRA	VRB	898
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
   src1 ← VSR[VRA+32].word[i]
   src2 ← VSR[VRB+32].word[i]

   lt_flag ← EXTS(src1) < EXTS(src2)
   VSR[VRT+32].word[i] ← lt_flag=1 ? src1 : src2
end
```

For each integer value i from 0 to 3, do the following.

Let src1 be the signed integer value in word element i of VSR[VRA+32].

Let src2 be the signed integer value in word element i of VSR[VRB+32].

src1 is compared to src2. The smaller of the two values is placed into word element i of VSR[VRT+32].

Special Registers Altered:

None

Vector Minimum Unsigned Word VX-form

vminuw VRT,VRA,VRB

4	VRT	VRA	VRB	642
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
   src1 ← VSR[VRA+32].word[i]
   src2 ← VSR[VRB+32].word[i]

   lt_flag ← EXTZ(src1) < EXTZ(src2)
   VSR[VRT+32].word[i] ← lt_flag=1 ? src1 : src2
end
```

For each integer value i from 0 to 3, do the following.

Let src1 be the unsigned integer value in word element i of VSR[VRA+32].

Let src2 be the unsigned integer value in word element i of VSR[VRB+32].

src1 is compared to src2. The smaller of the two values is placed into word element i of VSR[VRT+32].

Special Registers Altered:

None

Register Data Layout for vminsw & vminuw

src1	VSR[VRA+32].word[0]	VSR[VRA+32].word[1]	VSR[VRA+32].word[2]	VSR[VRA+32].word[3]
src2	VSR[VRB+32].word[0]	VSR[VRB+32].word[1]	VSR[VRB+32].word[2]	VSR[VRB+32].word[3]
result	VSR[VRT+32].word[0]	VSR[VRT+32].word[1]	VSR[VRT+32].word[2]	VSR[VRT+32].word[3]
	0	32	64	96 127

Vector Minimum Signed Doubleword VX-form

vminsd VRT,VRA,VRB

4	VRT	VRA	VRB	962
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 1
   src1 ← VSR[VRA+32].dword[i]
   src2 ← VSR[VRB+32].dword[i]

   lt_flag ← EXTS(src1) < EXTS(src2)
   VSR[VRT+32].dword[i] ← lt_flag=1 ? src1 : src2
end
```

For each integer value i from 0 to 1, do the following.

Let src1 be the signed integer value in doubleword element i of VSR[VRA+32].

Let src2 be the signed integer value in doubleword element i of VSR[VRB+32].

src1 is compared to src2. The smaller of the two values is placed into doubleword element i of VSR[VRT+32].

Special Registers Altered:

None

Vector Minimum Unsigned Doubleword VX-form

vminud VRT,VRA,VRB

4	VRT	VRA	VRB	706
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 1
   src1 ← VSR[VRA+32].dword[i]
   src2 ← VSR[VRB+32].dword[i]

   lt_flag ← EXTZ(src1) < EXTZ(src2)
   VSR[VRT+32].dword[i] ← lt_flag=1 ? src1 : src2
end
```

For each integer value i from 0 to 1, do the following.

Let src1 be the unsigned integer value in doubleword element i of VSR[VRA+32].

Let src2 be the unsigned integer value in doubleword element i of VSR[VRB+32].

src1 is compared to src2. The smaller of the two values is placed into doubleword element i of VSR[VRT+32].

Special Registers Altered:

None

Register Data Layout for vminsd & vminud

src1	VSR[VRA+32].dword[0]	VSR[VRA+32].dword[1]
src2	VSR[VRB+32].dword[0]	VSR[VRB+32].dword[1]
result	VSR[VRT+32].dword[0]	VSR[VRT+32].dword[1]
	0 64	127

6.9.3 Vector Integer Compare Instructions

The *Vector Integer Compare* instructions compare two VSRs element by element, interpreting the elements as unsigned or signed integers depending on the instruction, and set the corresponding element of the target VSR to all 1s if the relation being tested is true and to all 0s if the relation being tested is false.

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

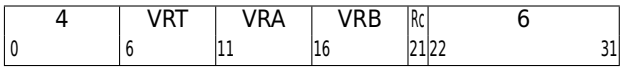
Bit	Description
0	The relation is true for all element pairs (i.e., `VSR[VRT+32]` is set to all 1s)
1	0
2	The relation is false for all element pairs (i.e., `VSR[VRT+32]` is set to all 0s)
3	0

Programming Note

***vcmpequb*[.], *vcmpequh*[.], *vcmpequw*[.], and *vcmpequd*[.]** can be used for unsigned or signed integers.

Vector Compare Equal Unsigned Byte VC-form

vcmpequb	VRT,VRA,VRB	(Rc=0)
vcmpequb.	VRT,VRA,VRB	(Rc=1)




```
if MSR.VEC=0 then Vector_Unavailable()

all_true  ← 1
all_false ← 1

do i = 0 to 15
   src1 ← VSR[VRA+32].byte[i]
   src2 ← VSR[VRB+32].byte[i]
   if src1 = src2 then do
      VSR[VRT+32].byte[i] ← 0xFF
      all_false ← 0
   end
   else do
      VSR[VRT+32].byte[i] ← 0x00
      all_true ← 0
   end
end

if Rc=1 then
   CR.field[6] ← all_true || 0b0 || all_false || 0b0
```

For each integer value `i` from 0 to 15, do the following.

Let `src1` be the unsigned integer value in byte element `i` of `VSR[VRA+32]`.

Let `src2` be the unsigned integer value in byte element `i` of `VSR[VRB+32]`.

`src1` is compared to `src2`.

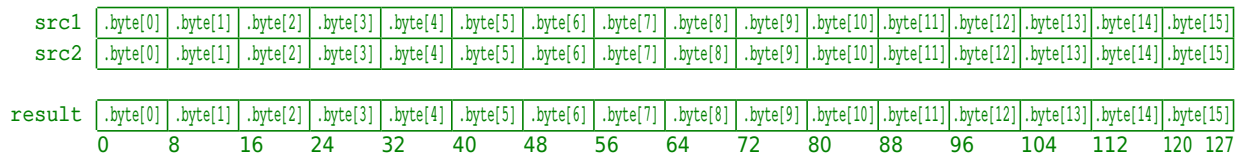
The contents of byte element `i` of `VSR[VRT+32]` are set to all 1s if `src1` is equal to `src2`, and is set to all 0s otherwise.

If `Rc=1`, CR field 6 is set to indicate whether all vector elements compared true and whether all vector elements compared false.

Special Registers Altered:

Register	*Field(s)*	
CR	CR6	(if Rc=1)

Register Data Layout for vcmpequb[.]



Vector Compare Equal Unsigned Halfword VC-form

vcmpequh	VRT,VRA,VRB	(Rc=0)
vcmpequh.	VRT,VRA,VRB	(Rc=1)

4	VRT	VRA	VRB	Rc	70
0	6	11	16	21	22 31

```
if MSR.VEC=0 then Vector_Unavailable()

all_true  ← 1
all_false ← 1

do i = 0 to 7
   src1 ← VSR[VRA+32].hword[i]
   src2 ← VSR[VRB+32].hword[i]
   if src1 = src2 then do
      VSR[VRT+32].hword[i] ← 0xFFFF
      all_false ← 0
   end
   else do
      VSR[VRT+32].hword[i] ← 0x0000
      all_true ← 0
   end
end

if Rc=1 then
   CR.field[6] ← all_true || 0b0 || all_false || 0b0
```

For each integer value `i` from 0 to 7, do the following.

Let `src1` be the unsigned integer value in halfword element `i` of `VSR[VRA+32]`.

Let `src2` be the unsigned integer value in halfword element `i` of `VSR[VRB+32]`.

`src1` is compared to `src2`.

The contents of halfword element `i` of `VSR[VRT+32]` are set to all 1s if `src1` is equal to `src2`, and is set to all 0s otherwise.

If `Rc=1`, CR field 6 is set to indicate whether all vector elements compared true and whether all vector elements compared false.

Special Registers Altered:

Register	*Field(s)*	
CR	CR6	(if Rc=1)

Register Data Layout for vcmpequh[.]

src1	VSR[VRA+32].hword[0]	VSR[VRA+32].hword[1]	VSR[VRA+32].hword[2]	VSR[VRA+32].hword[3]	VSR[VRA+32].hword[4]	VSR[VRA+32].hword[5]	VSR[VRA+32].hword[6]	VSR[VRA+32].hword[7]
src2	VSR[VRB+32].hword[0]	VSR[VRB+32].hword[1]	VSR[VRB+32].hword[2]	VSR[VRB+32].hword[3]	VSR[VRB+32].hword[4]	VSR[VRB+32].hword[5]	VSR[VRB+32].hword[6]	VSR[VRB+32].hword[7]

result	VSR[VRT+32].hword[0]	VSR[VRT+32].hword[1]	VSR[VRT+32].hword[2]	VSR[VRT+32].hword[3]	VSR[VRT+32].hword[4]	VSR[VRT+32].hword[5]	VSR[VRT+32].hword[6]	VSR[VRT+32].hword[7]
	0	16	32	48	64	80	96	112 127

Vector Compare Equal Unsigned Word VC-form

vcmpequw	VRT,VRA,VRB	(Rc=0)
vcmpequw.	VRT,VRA,VRB	(Rc=1)

4	VRT	VRA	VRB	Rc	134
0	6	11	16	21	22 31

```
if MSR.VEC=0 then Vector_Unavailable()

all_true  ← 1
all_false ← 1

do i = 0 to 3
   src1 ← VSR[VRA+32].word[i]
   src2 ← VSR[VRB+32].word[i]
   if src1 = src2 then do
      VSR[VRT+32].word[i] ← 0xFFFF_FFFF
      all_false ← 0
   end
   else do
      VSR[VRT+32].word[i] ← 0x0000_0000
      all_true ← 0
   end
end

if Rc=1 then
   CR.field[6] ← all_true || 0b0 || all_false || 0b0
```

For each integer value `i` from 0 to 3, do the following.

Let `src1` be the unsigned integer value in word element `i` of `VSR[VRA+32]`.

Let `src2` be the unsigned integer value in word element `i` of `VSR[VRB+32]`.

`src1` is compared to `src2`.

The contents of word element `i` of `VSR[VRT+32]` are set to all 1s if `src1` is equal to `src2`, and is set to all 0s otherwise.

If `Rc=1`, CR field 6 is set to indicate whether all vector elements compared true and whether all vector elements compared false.

Special Registers Altered:

Register	*Field(s)*	
CR	CR6	(if Rc=1)

Register Data Layout for vcmpequw[.]

src1	VSR[VRA+32].word[0]	VSR[VRA+32].word[1]	VSR[VRA+32].word[2]	VSR[VRA+32].word[3]
src2	VSR[VRB+32].word[0]	VSR[VRB+32].word[1]	VSR[VRB+32].word[2]	VSR[VRB+32].word[3]

result	VSR[VRT+32].word[0]	VSR[VRT+32].word[1]	VSR[VRT+32].word[2]	VSR[VRT+32].word[3]
	0	32	64	96 127

Vector Compare Equal Unsigned Doubleword VC-form

vcmpequd	VRT,VRA,VRB	(Rc=0)
vcmpequd.	VRT,VRA,VRB	(Rc=1)

4	VRT	VRA	VRB	Rc	199
0	6	11	16	21	22 31

```
if MSR.VEC=0 then Vector_Unavailable()

all_true  ← 1
all_false ← 1

do i = 0 to 1
   src1 ← VSR[VRA+32].dword[i]
   src2 ← VSR[VRB+32].dword[i]
   if src1 = src2 then do
      VSR[VRT+32].dword[i] ← 0xFFFF_FFFF_FFFF_FFFF
      all_false ← 0
   end
   else do
      VSR[VRT+32].dword[i] ← 0x0000_0000_0000_0000
      all_true ← 0
   end
end

if Rc=1 then
   CR.field[6] ← all_true || 0b0 || all_false || 0b0
```

For each integer value `i` from 0 to 1, do the following.

Let `src1` be the unsigned integer value in doubleword element `i` of `VSR[VRA+32]`.

Let `src2` be the unsigned integer value in doubleword element `i` of `VSR[VRB+32]`.

`src1` is compared to `src2`.

The contents of doubleword element `i` of `VSR[VRT+32]` are set to all 1s if `src1` is equal to `src2`, and is set to all 0s otherwise.

If `Rc=1`, CR field 6 is set to indicate whether all vector elements compared true and whether all vector elements compared false.

Special Registers Altered:

Register	*Field(s)*	
CR	CR6	(if Rc=1)

Register Data Layout for vcmpequd[.]

src1	VSR[VRA+32].dword[0]	VSR[VRA+32].dword[1]
src2	VSR[VRB+32].dword[0]	VSR[VRB+32].dword[1]
result	VSR[VRT+32].dword[0]	VSR[VRT+32].dword[1]
	0 64	127

Vector Compare Equal Quadword VC-form

vcmpequq	VRT,VRA,VRB	(Rc=0)
vcmpequq.	VRT,VRA,VRB	(Rc=1)

4	VRT	VRA	VRB	Rc	455
0	6	11	16	21	22 31

```
if MSR.VEC=0 then Vector_Unavailable()

all_true  ← 1
all_false ← 1

src1 ← VSR[VRA+32]
src2 ← VSR[VRB+32]

if src1 = src2 then do
   VSR[VRT+32] ←
      0xFFFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF
   all_false ← 0
end
else do
   VSR[VRT+32] ←
      0x0000_0000_0000_0000_0000_0000_0000_0000
   all_true ← 0
end

if Rc=1 then
   CR.field[6] ← all_true || 0b0 || all_false || 0b0
```

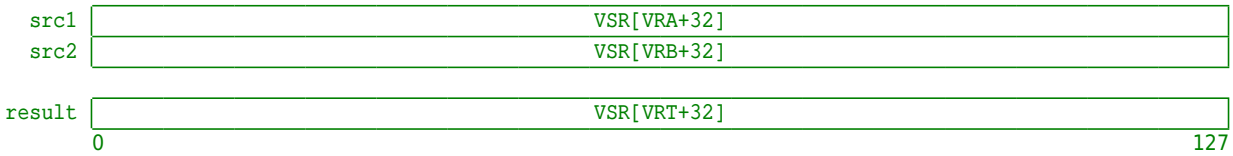
Let `src1` be the unsigned integer value in `VSR[VRA+32]`.
Let `src2` be the unsigned integer value in `VSR[VRB+32]`.

If `src1` is equal to `src2`, set `VSR[VRT+32]` to all 1s. Otherwise, set `VSR[VRT+32]` to all 0s.

Special Registers Altered:

Register	*Field(s)*	
CR	CR6	(if Rc=1)

Register Data Layout for vcmpequq[.]



Vector Compare Greater Than Signed Byte VC-form

vcmpgtsb	VRT,VRA,VRB	(Rc=0)
vcmpgtsb.	VRT,VRA,VRB	(Rc=1)

4	VRT	VRA	VRB	Rc	774
0	6	11	16	21	22 31

```
if MSR.VEC=0 then Vector_Unavailable()

all_true  ← 1
all_false ← 1

do i = 0 to 15
   src1 ← EXTS(VSR[VRA+32].byte[i])
   src2 ← EXTS(VSR[VRB+32].byte[i])

   if src1 > src2 then do
      VSR[VRT+32].byte[i] ← 0xFF
      all_false ← 0
   end
   else do
      VSR[VRT+32].byte[i] ← 0x00
      all_true ← 0
   end
end

if Rc=1 then
   CR.field[6] ← all_true || 0b0 || all_false || 0b0
```

For each integer value i from 0 to 15, do the following.

Let src1 be the unsigned integer value in byte element i of VSR[VRA+32].
Let src2 be the unsigned integer value in byte element i of VSR[VRB+32].

src1 is compared to src2.

The contents of byte element i of VSR[VRT+32] are set to all 1s if src1 is equal to src2, and is set to all 0s otherwise.

If Rc=1, CR field 6 is set to indicate whether all vector elements compared true and whether all vector elements compared false.

Special Registers Altered:

Register	*Field(s)*	
CR	CR6	(if Rc=1)

Vector Compare Greater Than Unsigned Byte VC-form

vcmpgtub	VRT,VRA,VRB	(Rc=0)
vcmpgtub.	VRT,VRA,VRB	(Rc=1)

4	VRT	VRA	VRB	Rc	518
0	6	11	16	21	22 31

```
if MSR.VEC=0 then Vector_Unavailable()

all_true  ← 1
all_false ← 1

do i = 0 to 15
   src1 ← EXTZ(VSR[VRA+32].byte[i])
   src2 ← EXTZ(VSR[VRB+32].byte[i])

   if src1 > src2 then do
      VSR[VRT+32].byte[i] ← 0xFF
      all_false ← 0
   end
   else do
      VSR[VRT+32].byte[i] ← 0x00
      all_true ← 0
   end
end

if Rc=1 then
   CR.field[6] ← all_true || 0b0 || all_false || 0b0
```

For each integer value i from 0 to 15, do the following.

Let src1 be the unsigned integer value in byte element i of VSR[VRA+32].
Let src2 be the unsigned integer value in byte element i of VSR[VRB+32].

src1 is compared to src2.

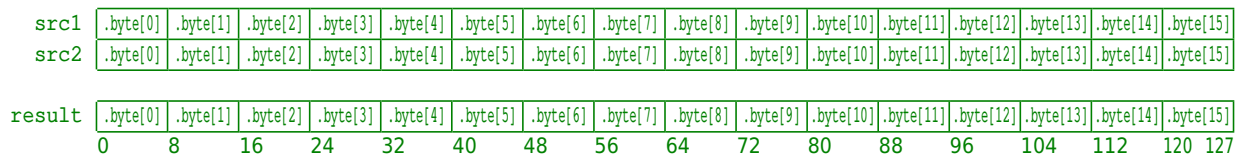
The contents of byte element i of VSR[VRT+32] are set to all 1s if src1 is greater than src2, and is set to all 0s otherwise.

If Rc=1, CR field 6 is set to indicate whether all vector elements compared true and whether all vector elements compared false.

Special Registers Altered:

Register	*Field(s)*	
CR	CR6	(if Rc=1)

Register Data Layout for vcmpgtsb[.] & vcmpgtub[.]



Vector Compare Greater Than Signed Halfword VC-form

vcmpgtsh	VRT,VRA,VRB	(Rc=0)
vcmpgtsh.	VRT,VRA,VRB	(Rc=1)

4	VRT	VRA	VRB	Rc	838
0	6	11	16	21	22 31

```
if MSR.VEC=0 then Vector_Unavailable()

all_true  ← 1
all_false ← 1

do i = 0 to 7
   src1 ← EXTS(VSR[VRA+32].hword[i])
   src2 ← EXTS(VSR[VRB+32].hword[i])

   if src1 > src2 then do
      VSR[VRT+32].hword[i] ← 0xFFFF
      all_false ← 0
   end
   else do
      VSR[VRT+32].hword[i] ← 0x0000
      all_true ← 0
   end
end

if Rc=1 then
   CR.field[6] ← all_true || 0b0 || all_false || 0b0
```

For each integer value i from 0 to 7, do the following.

Let src1 be the signed integer value in halfword element i of VSR[VRA+32].
Let src2 be the signed integer value in halfword element i of VSR[VRB+32].

src1 is compared to src2.

The contents of halfword element i of VSR[VRT+32] are set to all 1s if src1 is greater than src2, and is set to all 0s otherwise.

If Rc=1, CR field 6 is set to indicate whether all vector elements compared true and whether all vector elements compared false.

Special Registers Altered:

Register	*Field(s)*	
CR	CR6	(if Rc=1)

Vector Compare Greater Than Unsigned Halfword VC-form

vcmpgtuh	VRT,VRA,VRB	(Rc=0)
vcmpgtuh.	VRT,VRA,VRB	(Rc=1)

4	VRT	VRA	VRB	Rc	582
0	6	11	16	21	22 31

```
if MSR.VEC=0 then Vector_Unavailable()

all_true  ← 1
all_false ← 1

do i = 0 to 7
   src1 ← EXTZ(VSR[VRA+32].hword[i])
   src2 ← EXTZ(VSR[VRB+32].hword[i])

   if src1 > src2 then do
      VSR[VRT+32].hword[i] ← 0xFFFF
      all_false ← 0
   end
   else do
      VSR[VRT+32].hword[i] ← 0x0000
      all_true ← 0
   end
end

if Rc=1 then
   CR.field[6] ← all_true || 0b0 || all_false || 0b0
```

For each integer value i from 0 to 7, do the following.

Let src1 be the unsigned integer value in halfword element i of VSR[VRA+32].
Let src2 be the unsigned integer value in halfword element i of VSR[VRB+32].

src1 is compared to src2.

The contents of halfword element i of VSR[VRT+32] are set to all 1s if src1 is greater than src2, and is set to all 0s otherwise.

If Rc=1, CR field 6 is set to indicate whether all vector elements compared true and whether all vector elements compared false.

Special Registers Altered:

Register	*Field(s)*	
CR	CR6	(if Rc=1)

Register Data Layout for vcmpgtsh[.] & vcmpgtuh[.]

src1	VSR[VRA+32].hword[0]	VSR[VRA+32].hword[1]	VSR[VRA+32].hword[2]	VSR[VRA+32].hword[3]	VSR[VRA+32].hword[4]	VSR[VRA+32].hword[5]	VSR[VRA+32].hword[6]	VSR[VRA+32].hword[7]
src2	VSR[VRB+32].hword[0]	VSR[VRB+32].hword[1]	VSR[VRB+32].hword[2]	VSR[VRB+32].hword[3]	VSR[VRB+32].hword[4]	VSR[VRB+32].hword[5]	VSR[VRB+32].hword[6]	VSR[VRB+32].hword[7]

result	VSR[VRT+32].hword[0]	VSR[VRT+32].hword[1]	VSR[VRT+32].hword[2]	VSR[VRT+32].hword[3]	VSR[VRT+32].hword[4]	VSR[VRT+32].hword[5]	VSR[VRT+32].hword[6]	VSR[VRT+32].hword[7]
	0	16	32	48	64	80	96	112 127

Vector Compare Greater Than Signed Word VC-form

vcmpgtsw	VRT,VRA,VRB	(Rc=0)
vcmpgtsw.	VRT,VRA,VRB	(Rc=1)

4	VRT	VRA	VRB	Rc	902
0	6	11	16	21	22 31

```
if MSR.VEC=0 then Vector_Unavailable()

all_true  ← 1
all_false ← 1

do i = 0 to 3
   src1 ← EXTS(VSR[VRA+32].word[i])
   src2 ← EXTS(VSR[VRB+32].word[i])

   if src1 > src2 then do
      VSR[VRT+32].word[i] ← 0xFFFF_FFFF
      all_false ← 0
   end
   else do
      VSR[VRT+32].word[i] ← 0x0000_0000
      all_true ← 0
   end
end

if Rc=1 then
   CR.field[6] ← all_true || 0b0 || all_false || 0b0
```

For each integer value i from 0 to 3, do the following.

Let src1 be the signed integer value in word element i of VSR[VRA+32].
Let src2 be the signed integer value in word element i of VSR[VRB+32].

src1 is compared to src2.

The contents of word element i of VSR[VRT+32] are set to all 1s if src1 is greater than src2, and is set to all 0s otherwise.

If Rc=1, CR field 6 is set to indicate whether all vector elements compared true and whether all vector elements compared false.

Special Registers Altered:

Register	*Field(s)*	
CR	CR6	(if Rc=1)

Vector Compare Greater Than Unsigned Word VC-form

vcmpgtuw	VRT,VRA,VRB	(Rc=0)
vcmpgtuw.	VRT,VRA,VRB	(Rc=1)

4	VRT	VRA	VRB	Rc	646
0	6	11	16	21	22 31

```
if MSR.VEC=0 then Vector_Unavailable()

all_true  ← 1
all_false ← 1

do i = 0 to 3
   src1 ← EXTZ(VSR[VRA+32].word[i])
   src2 ← EXTZ(VSR[VRB+32].word[i])

   if src1 > src2 then do
      VSR[VRT+32].word[i] ← 0xFFFF_FFFF
      all_false ← 0
   end
   else do
      VSR[VRT+32].word[i] ← 0x0000_0000
      all_true ← 0
   end
end

if Rc=1 then
   CR.field[6] ← all_true || 0b0 || all_false || 0b0
```

For each integer value i from 0 to 3, do the following.

Let src1 be the unsigned integer value in word element i of VSR[VRA+32].
Let src2 be the unsigned integer value in word element i of VSR[VRB+32].

src1 is compared to src2.

The contents of word element i of VSR[VRT+32] are set to all 1s if src1 is greater than src2, and is set to all 0s otherwise.

If Rc=1, CR field 6 is set to indicate whether all vector elements compared true and whether all vector elements compared false.

Special Registers Altered:

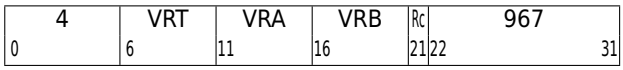
Register	*Field(s)*	
CR	CR6	(if Rc=1)

Register Data Layout for vcmpgtsw[.] & vcmpgtuw[.]

src1	VSR[VRA+32].word[0]	VSR[VRA+32].word[1]	VSR[VRA+32].word[2]	VSR[VRA+32].word[3]
src2	VSR[VRB+32].word[0]	VSR[VRB+32].word[1]	VSR[VRB+32].word[2]	VSR[VRB+32].word[3]
result	VSR[VRT+32].word[0]	VSR[VRT+32].word[1]	VSR[VRT+32].word[2]	VSR[VRT+32].word[3]
	0	32	64	96 127

Vector Compare Greater Than Signed Doubleword VC-form

vcmpgtsd	VRT,VRA,VRB	(Rc=0)
vcmpgtsd.	VRT,VRA,VRB	(Rc=1)




```
if MSR.VEC=0 then Vector_Unavailable()

all_true  ← 1
all_false ← 1

do i = 0 to 1
   src1 ← EXTS(VSR[VRA+32].dword[i])
   src2 ← EXTS(VSR[VRB+32].dword[i])

   if src1 > src2 then do
      VSR[VRT+32].dword[i] ← 0xFFFF_FFFF_FFFF_FFFF
      all_false ← 0
   end
   else do
      VSR[VRT+32].dword[i] ← 0x0000_0000_0000_0000
      all_true ← 0
   end
end

if Rc=1 then
   CR.field[6] ← all_true || 0b0 || all_false || 0b0
```

For each integer value `i` from 0 to 1, do the following.

Let `src1` be the signed integer value in doubleword element `i` of `VSR[VRA+32]`.
Let `src2` be the signed integer value in doubleword element `i` of `VSR[VRB+32]`.

`src1` is compared to `src2`.

The contents of doubleword element `i` of `VSR[VRT+32]` are set to all 1s if `src1` is greater than `src2`, and is set to all 0s otherwise.

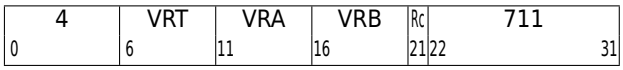
If `Rc=1`, CR field 6 is set to indicate whether all vector elements compared true and whether all vector elements compared false.

Special Registers Altered:

Register	*Field(s)*	
CR	CR6	(if Rc=1)

Vector Compare Greater Than Unsigned Doubleword VC-form

vcmpgtud	VRT,VRA,VRB	(Rc=0)
vcmpgtud.	VRT,VRA,VRB	(Rc=1)




```
if MSR.VEC=0 then Vector_Unavailable()

all_true  ← 1
all_false ← 1

do i = 0 to 1
   src1 ← EXTZ(VSR[VRA+32].dword[i])
   src2 ← EXTZ(VSR[VRB+32].dword[i])

   if src1 > src2 then do
      VSR[VRT+32].dword[i] ← 0xFFFF_FFFF_FFFF_FFFF
      all_false ← 0
   end
   else do
      VSR[VRT+32].dword[i] ← 0x0000_0000_0000_0000
      all_true ← 0
   end
end

if Rc=1 then
   CR.field[6] ← all_true || 0b0 || all_false || 0b0
```

For each integer value `i` from 0 to 1, do the following.

Let `src1` be the unsigned integer value in doubleword element `i` of `VSR[VRA+32]`.
Let `src2` be the unsigned integer value in doubleword element `i` of `VSR[VRB+32]`.

`src1` is compared to `src2`.

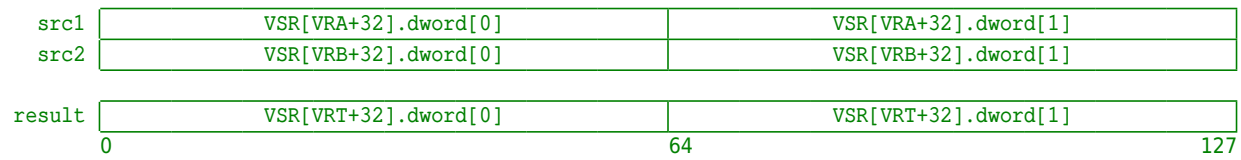
The contents of doubleword element `i` of `VSR[VRT+32]` are set to all 1s if `src1` is greater than `src2`, and is set to all 0s otherwise.

If `Rc=1`, CR field 6 is set to indicate whether all vector elements compared true and whether all vector elements compared false.

Special Registers Altered:

Register	*Field(s)*	
CR	CR6	(if Rc=1)

Register Data Layout for vcmpgtsd[.] & vcmpgtud[.]



Vector Compare Greater Than Signed Quadword VC-form

vcmpgtsq	VRT,VRA,VRB	(Rc=0)
vcmpgtsq.	VRT,VRA,VRB	(Rc=1)

4	VRT	VRA	VRB	Rc	903
0	6	11	16	21	22 31

```
if MSR.VEC=0 then Vector_Unavailable()

all_true  ← 1
all_false ← 1

src1 ← EXTS(VSR[VRA+32])
src2 ← EXTS(VSR[VRB+32])

if src1 > src2 then do
   VSR[VRT+32] ←
      0xFFFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF
   all_false ← 0
end
else do
   VSR[VRT+32] ←
      0x0000_0000_0000_0000_0000_0000_0000_0000
   all_true ← 0
end

if Rc=1 then
   CR.field[6] ← all_true || 0b0 || all_false || 0b0
```

Let src1 be the signed integer value in VSR[VRA+32].
Let src2 be the signed integer value in VSR[VRB+32].

src1 is compared to src2.

The contents of VSR[VRT+32] are set to all 1s if src1 is greater than src2, and are set to all 0s otherwise.

If Rc=1, CR field 6 is set to indicate if src1 is greater than src2 or if src1 is not greater than src2.

Special Registers Altered:

Register	*Field(s)*	
CR	CR6	(if Rc=1)

Vector Compare Greater Than Unsigned Quadword VC-form

vcmpgtuq	VRT,VRA,VRB	(Rc=0)
vcmpgtuq.	VRT,VRA,VRB	(Rc=1)

4	VRT	VRA	VRB	Rc	647
0	6	11	16	21	22 31

```
if MSR.VEC=0 then Vector_Unavailable()

all_true  ← 1
all_false ← 1

src1 ← EXTZ(VSR[VRA+32])
src2 ← EXTZ(VSR[VRB+32])

if src1 > src2 then do
   VSR[VRT+32] ←
      0xFFFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF
   all_false ← 0
end
else do
   VSR[VRT+32] ←
      0x0000_0000_0000_0000_0000_0000_0000_0000
   all_true ← 0
end

if Rc=1 then
   CR.field[6] ← all_true || 0b0 || all_false || 0b0
```

Let src1 be the unsigned integer value in VSR[VRA+32].
Let src2 be the unsigned integer value in VSR[VRB+32].

src1 is compared to src2.

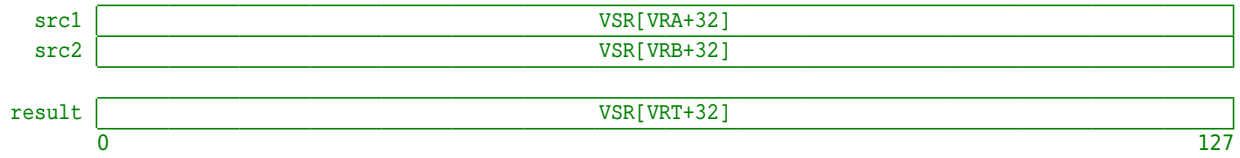
The contents of VSR[VRT+32] are set to all 1s if src1 is greater than src2, and are set to all 0s otherwise.

If Rc=1, CR field 6 is set to indicate if src1 is greater than src2 or if src1 is not greater than src2.

Special Registers Altered:

Register	*Field(s)*	
CR	CR6	(if Rc=1)

Register Data Layout for vcmpgtsq[.] & vcmpgtuq[.]



Vector Compare Not Equal Byte VC-form

vcmpneb	VRT,VRA,VRB	(Rc=0)
vcmpneb.	VRT,VRA,VRB	(Rc=1)

4	VRT	VRA	VRB	Rc	7
0	6	11	16	21	22 31

```
if MSR.VEC=0 then Vector_Unavailable()

all_true  ← 1
all_false ← 1

do i = 0 to 15
   src1 ← VSR[VRA+32].byte[i]
   src2 ← VSR[VRB+32].byte[i]

   if src1 != src2 then do
      VSR[VRT+32].byte[i] ← 0xFF
      all_false ← 0
   end
   else do
      VSR[VRT+32].byte[i] ← 0x00
      all_true ← 0
   end
end

if Rc=1 then
   CR.field[6] ← all_true || 0b0 || all_false || 0b0
```

For each integer value `i` from 0 to 15, do the following.

Let `src1` be the contents of byte element `i` of `VSR[VRA+32]`.
Let `src2` be the contents of byte element `i` of `VSR[VRB+32]`.

`src1` is compared to `src2`.

The contents of byte element `i` of `VSR[VRT+32]` are set to all 1s if `src1` is not equal to `src2`, and are set to all 0s otherwise.

If `Rc=1`, CR field 6 is set to indicate whether all vector elements compared true and whether all vector elements compared false.

Special Registers Altered:

Register	*Field(s)*	
CR	CR6	(if Rc=1)

Vector Compare Not Equal or Zero Byte VC-form

vcmpnezb	VRT,VRA,VRB	(Rc=0)
vcmpnezb.	VRT,VRA,VRB	(Rc=1)

4	VRT	VRA	VRB	Rc	263
0	6	11	16	21	22 31

```
if MSR.VEC=0 then Vector_Unavailable()

all_true  ← 1
all_false ← 1

do i = 0 to 15
   src1 ← VSR[VRA+32].byte[i]
   src2 ← VSR[VRB+32].byte[i]

   if src1 = 0 | src2 = 0 | src1 != src2 then do
      VSR[VRT+32].byte[i] ← 0xFF
      all_false ← 0
   end
   else do
      VSR[VRT+32].byte[i] ← 0x00
      all_true ← 0
   end
end

if Rc=1 then
   CR.field[6] ← all_true || 0b0 || all_false || 0b0
```

For each integer value `i` from 0 to 15, do the following.

Let `src1` be the contents of byte element `i` of `VSR[VRA+32]`.
Let `src2` be the contents of byte element `i` of `VSR[VRB+32]`.

`src1` is compared to `src2`.

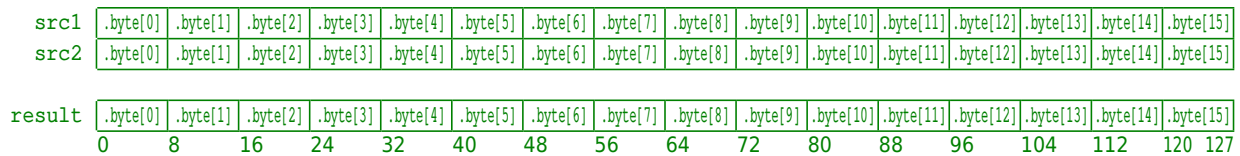
The contents of byte element `i` of `VSR[VRT+32]` are set to all 1s if `src1` is not equal to `src2` or either `src1` or `src2` is equal to `0x00`, and are set to all 0s otherwise.

If `Rc=1`, CR field 6 is set to indicate whether all vector elements compared true and whether all vector elements compared false.

Special Registers Altered:

Register	*Field(s)*	
CR	CR6	(if Rc=1)

Register Data Layout for vcmpneb[.] & vcmpnezb[.]



Vector Compare Not Equal Halfword VC-form

vcmpneh	VRT,VRA,VRB	(Rc=0)
vcmpneh.	VRT,VRA,VRB	(Rc=1)

4	VRT	VRA	VRB	Rc	71
0	6	11	16	21	22 31

```
if MSR.VEC=0 then Vector_Unavailable()

all_true  ← 1
all_false ← 1

do i = 0 to 7
   src1 ← VSR[VRA+32].hword[i]
   src2 ← VSR[VRB+32].hword[i]

   if src1 != src2 then do
      VSR[VRT+32].hword[i] ← 0xFFFF
      all_false ← 0
   end
   else do
      VSR[VRT+32].hword[i] ← 0x0000
      all_true ← 0
   end
end

if Rc=1 then
   CR.field[6] ← all_true || 0b0 || all_false || 0b0
```

For each integer value `i` from 0 to 7, do the following.

Let `src1` be the contents of halfword element `i` of `VSR[VRA+32]`.
Let `src2` be the contents of halfword element `i` of `VSR[VRB+32]`.

`src1` is compared to `src2`.

The contents of halfword element `i` of `VSR[VRT+32]` are set to all 1s if `src1` is not equal to `src2`, and is set to all 0s otherwise.

If `Rc=1`, CR field 6 is set to indicate whether all vector elements compared true and whether all vector elements compared false.

Special Registers Altered:

Register	*Field(s)*	
CR	CR6	(if Rc=1)

Vector Compare Not Equal or Zero Halfword VC-form

vcmpnezh	VRT,VRA,VRB	(Rc=0)
vcmpnezh.	VRT,VRA,VRB	(Rc=1)

4	VRT	VRA	VRB	Rc	327
0	6	11	16	21	22 31

```
if MSR.VEC=0 then Vector_Unavailable()

all_true  ← 1
all_false ← 1

do i = 0 to 7
   src1 ← VSR[VRA+32].hword[i]
   src2 ← VSR[VRB+32].hword[i]

   if src1 = 0 | src2 = 0 | src1 != src2 then do
      VSR[VRT+32].hword[i] ← 0xFFFF
      all_false ← 0
   end
   else do
      VSR[VRT+32].hword[i] ← 0x0000
      all_true ← 0
   end
end

if Rc=1 then
   CR.field[6] ← all_true || 0b0 || all_false || 0b0
```

For each integer value `i` from 0 to 7, do the following.

Let `src1` be the contents of halfword element `i` of `VSR[VRA+32]`.
Let `src2` be the contents of halfword element `i` of `VSR[VRB+32]`.

`src1` is compared to `src2`.

The contents of halfword element `i` of `VSR[VRT+32]` are set to all 1s if `src1` is not equal to `src2` or either `src1` or `src2` is equal to `0x00`, and is set to all 0s otherwise.

If `Rc=1`, CR field 6 is set to indicate whether all vector elements compared true and whether all vector elements compared false.

Special Registers Altered:

Register	*Field(s)*	
CR	CR6	(if Rc=1)

Register Data Layout for vcmpneh[.] & vcmpnezh[.]

src1	VSR[VRA+32].hword[0]	VSR[VRA+32].hword[1]	VSR[VRA+32].hword[2]	VSR[VRA+32].hword[3]	VSR[VRA+32].hword[4]	VSR[VRA+32].hword[5]	VSR[VRA+32].hword[6]	VSR[VRA+32].hword[7]
src2	VSR[VRB+32].hword[0]	VSR[VRB+32].hword[1]	VSR[VRB+32].hword[2]	VSR[VRB+32].hword[3]	VSR[VRB+32].hword[4]	VSR[VRB+32].hword[5]	VSR[VRB+32].hword[6]	VSR[VRB+32].hword[7]

result	VSR[VRT+32].hword[0]	VSR[VRT+32].hword[1]	VSR[VRT+32].hword[2]	VSR[VRT+32].hword[3]	VSR[VRT+32].hword[4]	VSR[VRT+32].hword[5]	VSR[VRT+32].hword[6]	VSR[VRT+32].hword[7]
	0	16	32	48	64	80	96	112 127

Vector Compare Not Equal Word VC-form

vcmpnew	VRT,VRA,VRB	(Rc=0)
vcmpnew.	VRT,VRA,VRB	(Rc=1)

4	VRT	VRA	VRB	Rc	135
0	6	11	16	21	22 31

```
if MSR.VEC=0 then Vector_Unavailable()

all_true  ← 1
all_false ← 1

do i = 0 to 3
   src1 ← VSR[VRA+32].word[i]
   src2 ← VSR[VRB+32].word[i]

   if src1 != src2 then do
      VSR[VRT+32].word[i] ← 0xFFFF_FFFF
      all_false ← 0
   end
   else do
      VSR[VRT+32].word[i] ← 0x0000_0000
      all_true ← 0
   end
end

if Rc=1 then
   CR.field[6] ← all_true || 0b0 || all_false || 0b0
```

For each integer value `i` from 0 to 3, do the following.

Let `src1` be the contents of word element `i` of `VSR[VRA+32]`.
Let `src2` be the contents of word element `i` of `VSR[VRB+32]`.

`src1` is compared to `src2`.

The contents of word element `i` of `VSR[VRT+32]` are set to all 1s if `src1` is not equal to `src2`, and is set to all 0s otherwise.

If `Rc=1`, CR field 6 is set to indicate whether all vector elements compared true and whether all vector elements compared false.

Special Registers Altered:

Register	*Field(s)*	
CR	CR6	(if Rc=1)

Vector Compare Not Equal or Zero Word VC-form

vcmpnezw	VRT,VRA,VRB	(Rc=0)
vcmpnezw.	VRT,VRA,VRB	(Rc=1)

4	VRT	VRA	VRB	Rc	391
0	6	11	16	21	22 31

```
if MSR.VEC=0 then Vector_Unavailable()

all_true  ← 1
all_false ← 1

do i = 0 to 3
   src1 ← VSR[VRA+32].word[i]
   src2 ← VSR[VRB+32].word[i]

   if src1 = 0 | src2 = 0 | src1 != src2 then do
      VSR[VRT+32].word[i] ← 0xFFFF_FFFF
      all_false ← 0
   end
   else do
      VSR[VRT+32].word[i] ← 0x0000_0000
      all_true ← 0
   end
end

if Rc=1 then
   CR.field[6] ← all_true || 0b0 || all_false || 0b0
```

For each integer value `i` from 0 to 3, do the following.

Let `src1` be the contents of word element `i` of `VSR[VRA+32]`.
Let `src2` be the contents of word element `i` of `VSR[VRB+32]`.

`src1` is compared to `src2`.

The contents of word element `i` of `VSR[VRT+32]` are set to all 1s if `src1` is not equal to `src2` or either `src1` or `src2` is equal to `0x00`, and is set to all 0s otherwise.

If `Rc=1`, CR field 6 is set to indicate whether all vector elements compared true and whether all vector elements compared false.

Special Registers Altered:

Register	*Field(s)*	
CR	CR6	(if Rc=1)

Register Data Layout for vcmpnew[.] & vcmpnezw[.]

src1	VSR[VRA+32].word[0]	VSR[VRA+32].word[1]	VSR[VRA+32].word[2]	VSR[VRA+32].word[3]
src2	VSR[VRB+32].word[0]	VSR[VRB+32].word[1]	VSR[VRB+32].word[2]	VSR[VRB+32].word[3]

result	VSR[VRT+32].word[0]	VSR[VRT+32].word[1]	VSR[VRT+32].word[2]	VSR[VRT+32].word[3]
	0	32	64	96 127

Vector Compare Signed Quadword VX-form

vcmpsq BF,VRA,VRB

4	BF	//	VRA	VRB	321
0	6	9	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

src1 ← EXTS(VSR[VRA+32])
src2 ← EXTS(VSR[VRB+32])

lt_flag ← src1 < src2
gt_flag ← src1 > src2
eq_flag ← src1 = src2

CR.field[BF] ← lt_flag<<3 | gt_flag<<2 | eq_flag<<1
```

Let src1 be the signed integer value in VSR[VRA+32].
Let src2 be the signed integer value in VSR[VRB+32].

Compare src1 with src2, place the comparison flags into CR field BF.

Special Registers Altered:

Register	*Field(s)*
CR	field BF

Vector Compare Unsigned Quadword VX-form

vcmpuq BF,VRA,VRB

4	BF	//	VRA	VRB	257
0	6	9	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

src1 ← EXTZ(VSR[VRA+32])
src2 ← EXTZ(VSR[VRB+32])

lt_flag ← src1 < src2
gt_flag ← src1 > src2
eq_flag ← src1 = src2

CR.field[BF] ← lt_flag<<3 | gt_flag<<2 | eq_flag<<1
```

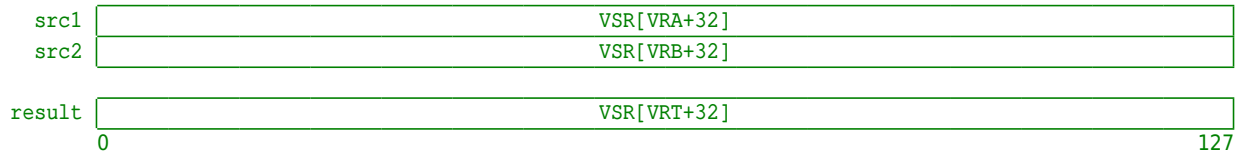
Let src1 be the unsigned integer value in VSR[VRA+32].
Let src2 be the unsigned integer value in VSR[VRB+32].

Compare src1 with src2, place the comparison flags into CR field BF.

Special Registers Altered:

Register	*Field(s)*
CR	field BF

Register Data Layout for vcmpsq & vcmpuq



6.9.4 Vector Logical Instructions

Extended mnemonics for vector logical operations

Extended mnemonics are provided that use the Vector OR and Vector NOR instructions to copy the contents of one VSR to another, with and without complementing. These are shown as examples with the two instructions.

Vector Move Register

Several vector instructions can be coded in a way such that they simply copy the contents of one VSR to another. An extended mnemonic is provided to convey the idea that no computation is being performed but merely data movement (from one register to another).

The following instruction copies the contents of register Vy to register Vx.

vmr Vx,Vy (equivalent to: vor Vx,Vy,Vy)

Vector Complement Register

The *Vector NOR* instruction can be coded in a way such that it complements the contents of one VSR and places the result into another VSR. An extended mnemonic is provided that allows this operation to be coded easily.

The following instruction complements the contents of register Vy and places the result into register Vx.

vnot Vx,Vy (equivalent to: vnor Vx,Vy,Vy)

Vector Logical AND VX-form

vand VRT,VRA,VRB

4	VRT	VRA	VRB	1028
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

VSR[VRT+32] ← VSR[VRA+32] & VSR[VRB+32]
```

The contents of VSR[VRA+32] are ANDed with the contents of VSR[VRB+32] and the result is placed into VSR[VRT+32].

Special Registers Altered:

None

Vector Logical AND with Complement VX-form

vandc VRT,VRA,VRB

4	VRT	VRA	VRB	1092
0	6	11	16	21 31

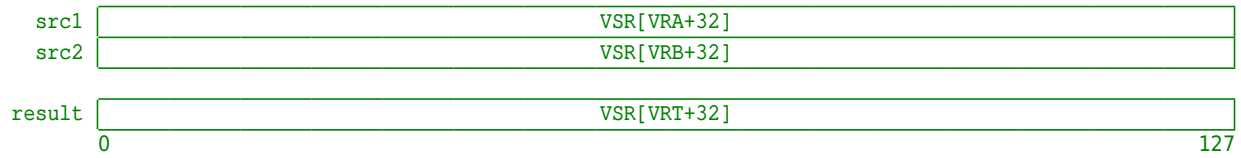
```
if MSR.VEC=0 then Vector_Unavailable()

VSR[VRT+32] ← VSR[VRA+32] & ¬VSR[VRB+32]
```

The contents of VSR[VRA+32] are ANDed with the complement of the contents of VSR[VRB+32] and the result is placed into VSR[VRT+32].

Special Registers Altered:

None

Register Data Layout for vand & vandc



Vector Logical Equivalence VX-form

veqv VRT,VRA,VRB

4	VRT	VRA	VRB	1668
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

VSR[VRT+32] ← VSR[VRA+32] ≡ VSR[VRB+32]
```

The contents of VSR[VRA+32] are XORed with the contents of VSR[VRB+32] and the complemented result is placed into VSR[VRT+32].

Special Registers Altered:

None

Vector Logical NAND VX-form

vnand VRT,VRA,VRB

4	VRT	VRA	VRB	1412
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

VSR[VRT+32] ← ¬( VSR[VRA+32] & VSR[VRB+32] )
```

The contents of VSR[VRA+32] are ANDed with the contents of VSR[VRB+32] and the complemented result is placed into VSR[VRT+32].

Special Registers Altered:

None

Vector Logical OR VX-form

vor VRT,VRA,VRB

4	VRT	VRA	VRB	1156
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

VSR[VRT+32] ← VSR[VRA+32] | VSR[VRB+32]
```

The contents of VSR[VRA+32] are ORed with the contents of VSR[VRB+32] and the result is placed into VSR[VRT+32].

Special Registers Altered:

None

Extended Mnemonics:

Extended Mnemonics for *Vector Logical OR*:

Extended mnemonic:		Equivalent to:	
vmr	VRT,vrs	vor	VRT,vrs,vrs

Vector Logical OR with Complement VX-form

vorc VRT,VRA,VRB

4	VRT	VRA	VRB	1348
0	6	11	16	21 31

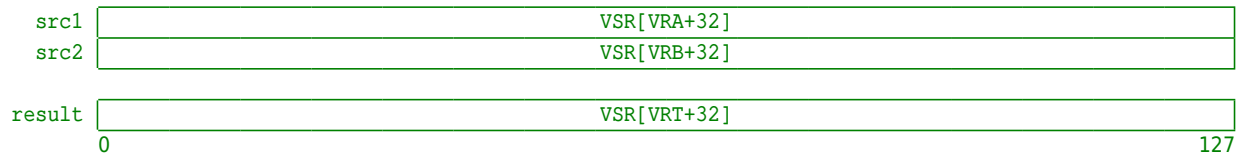
```
if MSR.VEC=0 then Vector_Unavailable()

VSR[VRT+32] ← VSR[VRA+32] | ¬VSR[VRB+32]
```

The contents of VSR[VRA+32] are ORed with the complement of the contents of VSR[VRB+32] and the result is placed into VSR[VRT+32].

Special Registers Altered:

None

Register Data Layout for veqv, vnand, vor & vorc



Vector Logical NOR VX-form

vnor VRT,VRA,VRB

4	VRT	VRA	VRB	1284
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

VSR[VRT+32] ← ¬( VSR[VRA+32] | VSR[VRB+32] )
```

The contents of VSR[VRA+32] are ORed with the contents of VSR[VRB+32] and the complemented result is placed into VSR[VRT+32].

Special Registers Altered:

None

Extended Mnemonics:

Extended Mnemonics for *Vector Logical NOR*:

Extended mnemonic:	Equivalent to:
vnot Vx,Vy	vnor Vx,Vy,Vy

Vector Logical XOR VX-form

vxor VRT,VRA,VRB

4	VRT	VRA	VRB	1220
0	6	11	16	21 31

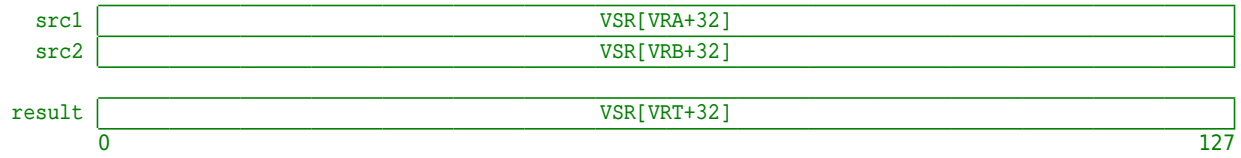
```
if MSR.VEC=0 then Vector_Unavailable()

VSR[VRT+32] ← VSR[VRA+32] ⊕ VSR[VRB+32]
```

The contents of VSR[VRA+32] are XORed with the contents of VSR[VRB+32] and the result is placed into VSR[VRT+32].

Special Registers Altered:

None

Register Data Layout for vnor & vxor

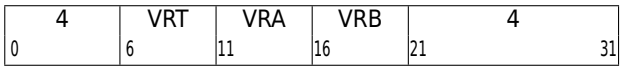


6.9.5 Vector Integer Rotate Instructions

6.9.5.1 Vector Integer Rotate Left Instructions

Vector Rotate Left Byte VX-form

vrlb VRT,VRA,VRB




```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 7
   src ← VSR[VRA+32].byte[i]
   sh  ← VSR[VRB+32].byte[i].bit[5:7]

   VSR[VRT+32].byte[i] ← src <<< sh
end
```

For each integer value `i` from 0 to 15, do the following.

Let `src1` be the contents of byte element `i` of `VSR[VRA+32]`.

Let `src2` be the contents of byte element `i` of `VSR[VRB+32]`.

`src1` is rotated left by the number of bits specified in the low-order 3 bits of `src2`.

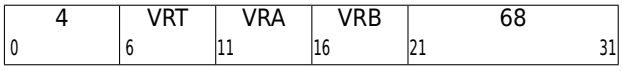
The result is placed into byte element `i` of `VSR[VRT+32]`.

Special Registers Altered:

None

Vector Rotate Left Halfword VX-form

vrlh VRT,VRA,VRB




```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 7
   src ← VSR[VRA+32].hword[i]
   sh  ← VSR[VRB+32].hword[i].bit[12:15]

   VSR[VRT+32].hword[i] ← src <<< sh
end
```

For each integer value `i` from 0 to 7, do the following.

Let `src1` be the contents of halfword element `i` of `VSR[VRA+32]`.

Let `src2` be the contents of halfword element `i` of `VSR[VRB+32]`.

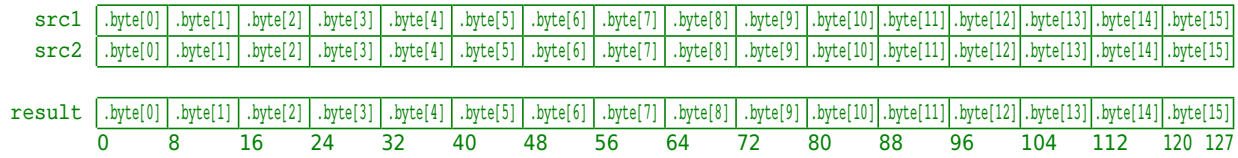
`src1` is rotated left by the number of bits specified in the low-order 4 bits of `src2`.

The result is placed into halfword element `i` of `VSR[VRT+32]`.

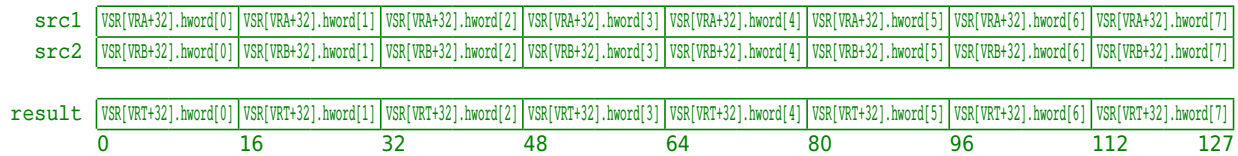
Special Registers Altered:

None

Register Data Layout for vrlb



Register Data Layout for vrlh



Vector Rotate Left Word VX-form

vrlw VRT,VRA,VRB

4	VRT	VRA	VRB	132
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
   src ← VSR[VRA+32].word[i]
   sh  ← VSR[VRB+32].word[i].bit[27:31]

   VSR[VRT+32].word[i] ← src <<< sh
end
```

For each integer value `i` from 0 to 3, do the following.

Let `src1` be the contents of word element `i` of `VSR[VRA+32]`.

Let `src2` be the contents of word element `i` of `VSR[VRB+32]`.

`src1` is rotated left by the number of bits specified in the low-order 5 bits of `src2`.

The result is placed into word element `i` in `VSR[VRT+32]`.

Special Registers Altered:

None

Vector Rotate Left Doubleword VX-form

vrld VRT,VRA,VRB

4	VRT	VRA	VRB	196
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 1
   src ← VSR[VRA+32].dword[i]
   sh  ← VSR[VRB+32].dword[i].bit[58:63]

   VSR[VRT+32].dword[i] ← src <<< sh
end
```

For each integer value `i` from 0 to 1, do the following.

Let `src1` be the contents of doubleword element `i` of `VSR[VRA+32]`.

Let `src2` be the contents of doubleword element `i` of `VSR[VRB+32]`.

`src1` is rotated left by the number of bits specified in the low-order 6 bits of `src2`.

The result is placed into doubleword element `i` of `VSR[VRT+32]`.

Special Registers Altered:

None

Register Data Layout for vrlw

src1	VSR[VRA+32].word[0]	VSR[VRA+32].word[1]	VSR[VRA+32].word[2]	VSR[VRA+32].word[3]
src2	VSR[VRB+32].word[0]	VSR[VRB+32].word[1]	VSR[VRB+32].word[2]	VSR[VRB+32].word[3]
result	VSR[VRT+32].word[0]	VSR[VRT+32].word[1]	VSR[VRT+32].word[2]	VSR[VRT+32].word[3]
	0	32	64	96 127

Register Data Layout for vrld

src1	VSR[VRA+32].dword[0]	VSR[VRA+32].dword[1]
src2	VSR[VRB+32].dword[0]	VSR[VRB+32].dword[1]
result	VSR[VRT+32].dword[0]	VSR[VRT+32].dword[1]
	0	64 127

Vector Rotate Left Quadword VX-form

vrlq VRT,VRA,VRB

4	VRT	VRA	VRB	5
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

n ← VSR[VRB+32].bit[57:63]
VSR[VRT+32] ← ROTL128(VSR[VRA+32], n)
```

Let SH be the contents of bits 57:63 of VSR[VRB+32].

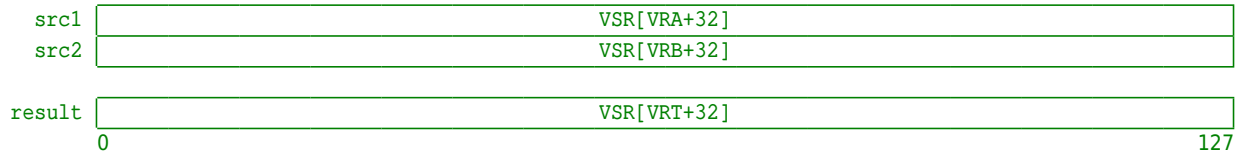
Let src1 be the contents of VSR[VRA+32].

src1 is rotated left by SH bits. Bits shifted out on the left are shifted in on the right to replace vacated bits.

Special Registers Altered:

None

Register Data Layout for vrlq



6.9.5.2 Vector Integer Rotate Left then AND with Mask Instructions

Vector Rotate Left Word then AND with Mask VX-form

vrlwnm VRT,VRA,VRB

4	VRT	VRA	VRB	389
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
   src1.word[0] ← VSR[VRA+32].word[i]
   src1.word[1] ← VSR[VRA+32].word[i]
   src2 ← VSR[VRB+32].word[i]
   b ← src2.bit[11:15]
   e ← src2.bit[19:23]
   n ← src2.bit[27:31]
   r ← src1.bit[n:n+31]
   m ← MASK(b, e)
   VSR[VRT+32].word[i] ← r & m
```

For each integer value `i` from 0 to 3, do the following.

Let `src1` be the contents of word element `i` of `VSR[VRA+32]`.
Let `src2` be the contents of word element `i` of `VSR[VRB+32]`.
Let `mb` be the contents of bits `11:15` of `src2`.
Let `me` be the contents of bits `19:23` of `src2`.
Let `sh` be the contents of bits `27:31` of `src2`.

`src1` is rotated left `sh` bits.

A mask is generated having 1-bits from bit `mb` through bit `me` and 0-bits elsewhere.

The rotated data are ANDed with the generated mask.

The result is placed into word element `i` of `VSR[VRT+32]`.

Special Registers Altered:

None

Vector Rotate Left Doubleword then AND with Mask VX-form

vrldnm VRT,VRA,VRB

4	VRT	VRA	VRB	453
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 1
   src1.dword[0] ← VSR[VRA+32].dword[i]
   src1.dword[1] ← VSR[VRA+32].dword[i]
   src2 ← VSR[VRB+32].dword[i]
   b ← src2.bit[42:47]
   e ← src2.bit[50:55]
   n ← src2.bit[58:63]
   r ← src1.bit[n:n+63]
   m ← MASK(b, e)
   VSR[VRT+32].dword[i] ← r & m
```

For each integer value `i` from 0 to 1, do the following.

Let `src1` be the contents of doubleword element `i` of `VSR[VRA+32]`.
Let `src2` be the contents of doubleword element `i` of `VSR[VRB+32]`.
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 `VSR[VRT+32]`.

Special Registers Altered:

None

Register Data Layout for vrlwnm

src1	VSR[VRA+32].word[0]	VSR[VRA+32].word[1]	VSR[VRA+32].word[2]	VSR[VRA+32].word[3]
src2	VSR[VRB+32].word[0]	VSR[VRB+32].word[1]	VSR[VRB+32].word[2]	VSR[VRB+32].word[3]
result	VSR[VRT+32].word[0]	VSR[VRT+32].word[1]	VSR[VRT+32].word[2]	VSR[VRT+32].word[3]
	0	32	64	96 127

Register Data Layout for vrldnm

src1	VSR[VRA+32].dword[0]	VSR[VRA+32].dword[1]
src2	VSR[VRB+32].dword[0]	VSR[VRB+32].dword[1]
result	VSR[VRT+32].dword[0]	VSR[VRT+32].dword[1]
	0	64 127

Vector Rotate Left Quadword then AND with Mask VX-form

vrlqnm VRT,VRA,VRB

4	VRT	VRA	VRB	325
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

b ← VSR[VRB+32].bit[41:47]
e ← VSR[VRB+32].bit[49:55]
n ← VSR[VRB+32].bit[57:63]
r ← ROTL128(VSR[VRA+32],n)
m ← MASK128(b, e)
VSR[VRT+32] ← r & m
```

Let `src1` be the contents of `VSR[VRA+32]`.
Let `src2` be the contents of `VSR[VRB+32]`.
Let `mb` be the contents of bits 41:47 of `src2`.
Let `me` be the contents of bits 49:55 of `src2`.
Let `sh` be the contents of bits 57:63 of `src2`.

`src1` is rotated left `sh` bits.

A mask is generated having 1-bits from bit `mb` through bit `me` and 0-bits elsewhere.

The rotated data are ANDed with the generated mask.

The result is placed into `VSR[VRT+32]`.

Special Registers Altered:

None

Register Data Layout for vrlqnm

6.9.5.3 Vector Integer Rotate Left then Mask Insert Instructions

Vector Rotate Left Word then Mask Insert VX-form

vrlwmi VRT,VRA,VRB

4	VRT	VRA	VRB	133
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
   src1.word[0] ← VSR[VRA+32].word[i]
   src1.word[1] ← VSR[VRA+32].word[i]
   src2 ← VSR[VRB+32].word[i]
   src3 ← VSR[VRT+32].word[i]
   b ← src2.bit[11:15]
   e ← src2.bit[19:23]
   n ← src2.bit[27:31]
   r ← src1.bit[n:n+31]
   m ← MASK(b, e)
   VSR[VRT+32].word[i] ← (r & m) | (src3 & ¬m)
```

For each integer value `i` from 0 to 3, do the following.

Let `src1` be the contents of word element `i` of `VSR[VRA+32]`.
Let `src2` be the contents of word element `i` of `VSR[VRB+32]`.
Let `src3` be the contents of word element `i` of `VSR[VRT+32]`.
Let `mb` be the contents of bits `11:15` of `src2`.
Let `me` be the contents of bits `19:23` of `src2`.
Let `sh` be the contents of bits `27:31` of `src2`.

`src1` is rotated left `sh` bits.

A mask is generated having 1-bits from bit `mb` through bit `me` and 0-bits elsewhere.

The rotated data are inserted into `src3` under control of the generated mask.

The result is placed into word element `i` of `VSR[VRT+32]`.

Special Registers Altered:

None

Register Data Layout for vrlwmi

src1	VSR[VRA+32].word[0]	VSR[VRA+32].word[1]	VSR[VRA+32].word[2]	VSR[VRA+32].word[3]
src2	VSR[VRB+32].word[0]	VSR[VRB+32].word[1]	VSR[VRB+32].word[2]	VSR[VRB+32].word[3]
result	VSR[VRT+32].word[0]	VSR[VRT+32].word[1]	VSR[VRT+32].word[2]	VSR[VRT+32].word[3]
	0	32	64	96 127

Vector Rotate Left Doubleword then Mask Insert VX-form

vrldmi VRT,VRA,VRB

4	VRT	VRA	VRB	197
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 1
   src1.dword[0] ← VSR[VRA+32].dword[i]
   src1.dword[1] ← VSR[VRA+32].dword[i]
   src2 ← VSR[VRB+32].dword[i]
   src3 ← VSR[VRT+32].dword[i]
   b ← src2.bit[42:47]
   e ← src2.bit[50:55]
   n ← src2.bit[58:63]
   r ← src1.bit[n:n+63]
   m ← MASK(b, e)
   VSR[VRT+32].dword[i] ← (r & m) | (src3 & ¬m)
```

For each integer value i from 0 to 1, do the following.

Let src1 be the contents of doubleword element i of VSR[VRA+32].
Let src2 be the contents of doubleword element i of VSR[VRB+32].
Let src3 be the contents of doubleword element i of VSR[VRT+32].
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 VSR[VRT+32].

Special Registers Altered:

None

Vector Rotate Left Quadword then Mask Insert VX-form

vrlqmi VRT,VRA,VRB

4	VRT	VRA	VRB	69
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

b ← VSR[VRB+32].bit[41:47]
e ← VSR[VRB+32].bit[49:55]
n ← VSR[VRB+32].bit[57:63]
r ← ROTL128(VSR[VRA+32],n)
m ← MASK128(b, e)
VSR[VRT+32] ← (r & m) | (VSR[VRT+32] & ¬m)
```

Let src1 be the contents of VSR[VRA+32].
Let src2 be the contents of VSR[VRB+32].
Let src3 be the contents of VSR[VRT+32].
Let mb be the contents of bits 41:47 of src2.
Let me be the contents of bits 49:55 of src2.
Let sh be the contents of bits 57:63 of src2.

src1 is rotated left sh bits.

A mask is generated having 1-bits from bit mb through bit me and 0-bits elsewhere.

The rotated data are inserted into src3 under control of the generated mask.

The result is placed into VSR[VRT+32].

Special Registers Altered:

None

Register Data Layout for vrldmi

src1	VSR[VRA+32].dword[0]	VSR[VRA+32].dword[1]
src2	VSR[VRB+32].dword[0]	VSR[VRB+32].dword[1]
result	VSR[VRT+32].dword[0]	VSR[VRT+32].dword[1]
	0	64 127

Register Data Layout for vrlqmi

src1	VSR[VRA+32]
src2	VSR[VRB+32]
result	VSR[VRT+32]
	0 127

6.9.6 Vector Integer Shift Instructions

6.9.6.1 Vector Integer Shift Left Instructions

Vector Shift Left Byte VX-form

vslb VRT,VRA,VRB

4	VRT	VRA	VRB	260
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 15
   src1 ← VSR[VRA+32].byte[i]
   src2 ← VSR[VRB+32].byte[i].bit[5:7]
   VSR[VRT+32].byte[i] ← src1 << src2
end
```

For each integer value `i` from 0 to 15, do the following.

Let `src1` be the contents of byte element `i` of `VSR[VRA+32]`.

Let `src2` be the contents of byte element `i` of `VSR[VRB+32]`.

`src1` is shifted left by the number of bits specified in the low-order 3 bits of `src2`.

- Bits shifted out the most-significant bit are lost.
- Zeros are supplied to the vacated bits on the right.

The result is placed into byte element `i` of `VSR[VRT+32]`.

Special Registers Altered:

None

Vector Shift Left Halfword VX-form

vslh VRT,VRA,VRB

4	VRT	VRA	VRB	324
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 7
   src1 ← VSR[VRA+32].hword[i]
   src2 ← VSR[VRB+32].hword[i].bit[12:15]
   VSR[VRT+32].hword[i] ← src1 << src2
end
```

For each integer value `i` from 0 to 7, do the following.

Let `src1` be the contents of halfword element `i` of `VSR[VRA+32]`.

Let `src2` be the contents of halfword element `i` of `VSR[VRB+32]`.

`src1` is shifted left by the number of bits specified in the low-order 4 bits of `src2`.

- Bits shifted out the most-significant bit are lost.
- Zeros are supplied to the vacated bits on the right.

The result is placed into halfword element `i` of `VSR[VRT+32]`.

Special Registers Altered:

None

Register Data Layout for vslb

src1	.byte[0]	.byte[1]	.byte[2]	.byte[3]	.byte[4]	.byte[5]	.byte[6]	.byte[7]	.byte[8]	.byte[9]	.byte[10]	.byte[11]	.byte[12]	.byte[13]	.byte[14]	.byte[15]
src2	.byte[0]	.byte[1]	.byte[2]	.byte[3]	.byte[4]	.byte[5]	.byte[6]	.byte[7]	.byte[8]	.byte[9]	.byte[10]	.byte[11]	.byte[12]	.byte[13]	.byte[14]	.byte[15]
result	.byte[0]	.byte[1]	.byte[2]	.byte[3]	.byte[4]	.byte[5]	.byte[6]	.byte[7]	.byte[8]	.byte[9]	.byte[10]	.byte[11]	.byte[12]	.byte[13]	.byte[14]	.byte[15]
	0	8	16	24	32	40	48	56	64	72	80	88	96	104	112	120 127

Register Data Layout for vslh

src1	VSR[VRA+32].hword[0]	VSR[VRA+32].hword[1]	VSR[VRA+32].hword[2]	VSR[VRA+32].hword[3]	VSR[VRA+32].hword[4]	VSR[VRA+32].hword[5]	VSR[VRA+32].hword[6]	VSR[VRA+32].hword[7]
src2	VSR[VRB+32].hword[0]	VSR[VRB+32].hword[1]	VSR[VRB+32].hword[2]	VSR[VRB+32].hword[3]	VSR[VRB+32].hword[4]	VSR[VRB+32].hword[5]	VSR[VRB+32].hword[6]	VSR[VRB+32].hword[7]
result	VSR[VRT+32].hword[0]	VSR[VRT+32].hword[1]	VSR[VRT+32].hword[2]	VSR[VRT+32].hword[3]	VSR[VRT+32].hword[4]	VSR[VRT+32].hword[5]	VSR[VRT+32].hword[6]	VSR[VRT+32].hword[7]
	0	16	32	48	64	80	96	112 127

Vector Shift Left Word VX-form

vslw VRT,VRA,VRB

4	VRT	VRA	VRB	388
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
   src1 ← VSR[VRA+32].word[i]
   src2 ← VSR[VRB+32].word[i].bit[27:31]

   VSR[VRT+32].word[i] ← src1 << src2
end
```

For each integer value `i` from 0 to 3, do the following.

Let `src1` be the contents of word element `i` of `VSR[VRA+32]`.

Let `src2` be the contents of word element `i` of `VSR[VRB+32]`.

`src1` is shifted left by the number of bits specified in the low-order 5 bits of `src2`.

- Bits shifted out the most-significant bit are lost.
- Zeros are supplied to the vacated bits on the right.

The result is placed into word element `i` of `VSR[VRT+32]`.

Special Registers Altered:

None

Vector Shift Left Doubleword VX-form

vsld VRT,VRA,VRB

4	VRT	VRA	VRB	1476
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 1
   src1 ← VSR[VRA+32].dword[i]
   src2 ← VSR[VRB+32].dword[i].bit[58:63]

   VSR[VRT+32].dword[i] ← src1 << src2
end
```

For each integer value `i` from 0 to 1, do the following.

Let `src1` be the contents of doubleword element `i` of `VSR[VRA+32]`.

Let `src2` be the contents of doubleword element `i` of `VSR[VRB+32]`.

`src1` is shifted left by the number of bits specified in the low-order 6 bits of `src2`.

- Bits shifted out the most-significant bit are lost.
- Zeros are supplied to the vacated bits on the right.

The result is placed into doubleword element `i` of `VSR[VRT+32]`.

Special Registers Altered:

None

Register Data Layout for vslw

src1	VSR[VRA+32].word[0]	VSR[VRA+32].word[1]	VSR[VRA+32].word[2]	VSR[VRA+32].word[3]
src2	VSR[VRB+32].word[0]	VSR[VRB+32].word[1]	VSR[VRB+32].word[2]	VSR[VRB+32].word[3]
result	VSR[VRT+32].word[0]	VSR[VRT+32].word[1]	VSR[VRT+32].word[2]	VSR[VRT+32].word[3]
	0	32	64	96 127

Register Data Layout for vsld

src1	VSR[VRA+32].dword[0]	VSR[VRA+32].dword[1]
src2	VSR[VRB+32].dword[0]	VSR[VRB+32].dword[1]
result	VSR[VRT+32].dword[0]	VSR[VRT+32].dword[1]
	0	64 127

Vector Shift Left Quadword VX-form

vslq VRT,VRA,VRB

4	VRT	VRA	VRB	261
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

VSR[VRT+32] ← VSR[VRA+32] << VSR[VRB+32].bit[57:63]
```

Let n be the contents of bits 57:63 of VSR[VRB+32].
Let src1 be the contents of VSR[VRA+32].
Let src2 be the contents of VSR[VRB+32].

src1 is shifted left by the number of bits specified in the low-order 7 bits of src2.

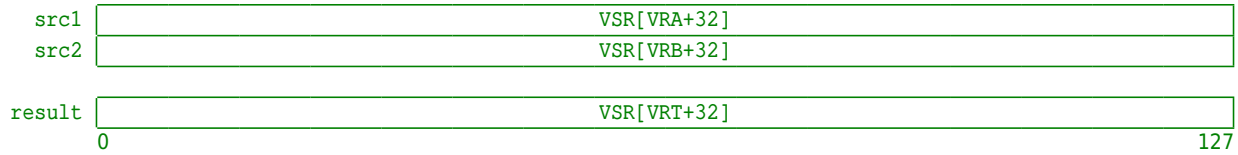
- Bits shifted out the most-significant bit are lost.
- Zeros are supplied to the vacated bits on the right.

The result is placed into VSR[VRT+32].

Special Registers Altered:

None

Register Data Layout for vslq



6.9.6.2 Vector Integer Shift Right Instructions

Vector Shift Right Byte VX-form

vsrb VRT,VRA,VRB

4	VRT	VRA	VRB	516
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 15
   src1 ← VSR[VRA+32].byte[i]
   src2 ← VSR[VRB+32].byte[i].bit[5:7]
   VSR[VRT+32].byte[i] ← CHOP8(EXTZ(src1) >> src2)
end
```

For each integer value `i` from 0 to 15, do the following.

Let `src1` be the contents of byte element `i` of `VSR[VRA+32]`.
Let `src2` be the contents of byte element `i` of `VSR[VRB+32]`.

`src1` is shifted right by the number of bits specified in the low-order 3 bits of `src2`.

- Bits shifted out the least-significant bit are lost.
- Zeros are supplied to the vacated bits on the left.

The result is placed into byte element `i` of `VSR[VRT+32]`.

Special Registers Altered:

None

Vector Shift Right Halfword VX-form

vsrh VRT,VRA,VRB

4	VRT	VRA	VRB	580
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 7
   src ← VSR[VRA+32].hword[i]
   src2 ← VSR[VRB+32].hword[i].bit[12:15]
   VSR[VRT+32].hword[i] ← CHOP16(EXTZ(src1) >> src2)
end
```

For each integer value `i` from 0 to 7, do the following.

Let `src1` be the contents of halfword element `i` of `VSR[VRA+32]`.
Let `src2` be the contents of halfword element `i` of `VSR[VRB+32]`.

`src1` is shifted right by the number of bits specified in the low-order 4 bits of `src2`.

- Bits shifted out the least-significant bit are lost.
- Zeros are supplied to the vacated bits on the left.

The result is placed into halfword element `i` of `VSR[VRT+32]`.

Special Registers Altered:

None

Register Data Layout for vsrb

	.byte[0]	.byte[1]	.byte[2]	.byte[3]	.byte[4]	.byte[5]	.byte[6]	.byte[7]	.byte[8]	.byte[9]	.byte[10]	.byte[11]	.byte[12]	.byte[13]	.byte[14]	.byte[15]
src1																
src2																
result																

0 8 16 24 32 40 48 56 64 72 80 88 96 104 112 120 127

Register Data Layout for vsrh

	VSR[VRA+32].hword[0]	VSR[VRA+32].hword[1]	VSR[VRA+32].hword[2]	VSR[VRA+32].hword[3]	VSR[VRA+32].hword[4]	VSR[VRA+32].hword[5]	VSR[VRA+32].hword[6]	VSR[VRA+32].hword[7]
src1								
src2								
result								

0 16 32 48 64 80 96 112 127

Vector Shift Right Word VX-form

vsrw VRT,VRA,VRB

4	VRT	VRA	VRB	644
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
   src1 ← VSR[VRA+32].word[i]
   src2 ← VSR[VRB+32].word[i].bit[27:31]
   VSR[VRT+32].word[i] ← CHOP32(EXTZ(src1) >> src2)
end
```

For each integer value `i` from 0 to 3, do the following.

Let `src1` be the contents of word element `i` of `VSR[VRA+32]`.
Let `src2` be the contents of word element `i` of `VSR[VRB+32]`.

`src1` is shifted right by the number of bits specified in the low-order 5 bits of `src2`.

- Bits shifted out the least-significant bit are lost.
- Zeros are supplied to the vacated bits on the left.

The result is placed into word element `i` of `VSR[VRT+32]`.

Special Registers Altered:

None

Vector Shift Right Doubleword VX-form

vsrd VRT,VRA,VRB

4	VRT	VRA	VRB	1732
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 1
   src1 ← VSR[VRA+32].dword[i]
src2 ← VSR[VRB+32].dword[i].bit[58:63]
   VSR[VRT+32].dword[i] ← CHOP64(EXTZ(src1) >> src2)
end
```

For each integer value `i` from 0 to 1, do the following.

Let `src1` be the contents of doubleword element `i` of `VSR[VRA+32]`.

Let `src2` be the contents of doubleword element `i` of `VSR[VRB+32]`.

`src1` are shifted right by the number of bits specified in bits 58:63 of `src2`.

- Bits shifted out the least-significant bit are lost.
- Zeros are supplied to the vacated bits on the left.

The result is placed into doubleword element `i` of `VSR[VRT+32]`.

Special Registers Altered:

None

Register Data Layout for vsrw

src1	VSR[VRA+32].word[0]	VSR[VRA+32].word[1]	VSR[VRA+32].word[2]	VSR[VRA+32].word[3]
src2	VSR[VRB+32].word[0]	VSR[VRB+32].word[1]	VSR[VRB+32].word[2]	VSR[VRB+32].word[3]
result	VSR[VRT+32].word[0]	VSR[VRT+32].word[1]	VSR[VRT+32].word[2]	VSR[VRT+32].word[3]
	0	32	64	96 127

Register Data Layout for vsrd

src1	VSR[VRA+32].dword[0]	VSR[VRA+32].dword[1]
src2	VSR[VRB+32].dword[0]	VSR[VRB+32].dword[1]
result	VSR[VRT+32].dword[0]	VSR[VRT+32].dword[1]
	0	64 127

Vector Shift Right Quadword VX-form

vsrq VRT,VRA,VRB

4	VRT	VRA	VRB	517
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

src1 ← VSR[VRA+32]
src2 ← VSR[VRB+32].bit[57:63]
VSR[VRT+32] ← CHOP128(EXTZ(src1) >> src2)
```

Let `src1` be the contents of `VSR[VRA+32]`.
Let `src2` be the contents of `VSR[VRB+32]`.

`src1` is shifted right by the number of bits specified in the low-order 7 bits of `src2`.

- Bits shifted out the least-significant bit are lost.
- Zeros are supplied to the vacated bits on the left.

The result is placed into `VSR[VRT+32]`.

Special Registers Altered:

None

Register Data Layout for vsrq

6.9.6.3 Vector Integer Shift Right Algebraic Instructions

Vector Shift Right Algebraic Byte VX-form

vsrab VRT,VRA,VRB

4	VRT	VRA	VRB	772
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 15
   src1 ← VSR[VRA+32].byte[i]
   src2 ← VSR[VRB+32].byte[i].bit[5:7]
   VSR[VRT+32].byte[i] ← CHOP8(EXTS(src1) >> src2)
end
```

For each integer value `i` from 0 to 15, do the following.

Let `src1` be the contents of byte element `i` of `VSR[VRA+32]`.
Let `src2` be the contents of byte element `i` of `VSR[VRB+32]`.

`src1` is shifted right by the number of bits specified in the low-order 3 bits of `src2`.

- Bits shifted out the least-significant bit are lost.
- Copies of bit 0 of `src1` are supplied to the vacated bits on the left.

The result is placed into byte element `i` of `VSR[VRT+32]`.

Special Registers Altered:

None

Vector Shift Right Algebraic Halfword VX-form

vsrah VRT,VRA,VRB

4	VRT	VRA	VRB	836
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 7
   src1 ← VSR[VRA+32].hword[i]
   src2 ← VSR[VRB+32].hword[i].bit[12:15]
   VSR[VRT+32].hword[i] ← CHOP16(EXTS(src1) >> src2)
end
```

For each integer value `i` from 0 to 7, do the following.

Let `src1` be the contents of halfword element `i` of `VSR[VRA+32]`.
Let `src2` be the contents of halfword element `i` of `VSR[VRB+32]`.

`src1` is shifted right by the number of bits specified in the low-order 4 bits of `src2`.

- Bits shifted out the least-significant bit are lost.
- Copies of bit 0 of `src1` are supplied to the vacated bits on the left.

The result is placed into halfword element `i` of `VSR[VRT+32]`.

Special Registers Altered:

None

Register Data Layout for vsrab

	.byte[0]	.byte[1]	.byte[2]	.byte[3]	.byte[4]	.byte[5]	.byte[6]	.byte[7]	.byte[8]	.byte[9]	.byte[10]	.byte[11]	.byte[12]	.byte[13]	.byte[14]	.byte[15]
src1																
src2																
result																

0 8 16 24 32 40 48 56 64 72 80 88 96 104 112 120 127

Register Data Layout for vsrah

	VSR[VRA+32].hword[0]	VSR[VRA+32].hword[1]	VSR[VRA+32].hword[2]	VSR[VRA+32].hword[3]	VSR[VRA+32].hword[4]	VSR[VRA+32].hword[5]	VSR[VRA+32].hword[6]	VSR[VRA+32].hword[7]
src1								
src2								
result								

0 16 32 48 64 80 96 112 127

Vector Shift Right Algebraic Word VX-form

vsraw VRT,VRA,VRB

4	VRT	VRA	VRB	900
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
   src1 ← VSR[VRA+32].word[i]
   src2 ← VSR[VRB+32].word[i].bit[27:31]
   VSR[VRT+32].word[i] ← CHOP32(EXTS(src1) >> src2)
end
```

For each integer value i from 0 to 3, do the following.

Let src1 be the contents of word element i of VSR[VRA+32].
Let src2 be the contents of word element i of VSR[VRB+32].

src1 is shifted right by the number of bits specified in the low-order 5 bits of src2.

- Bits shifted out the least-significant bit are lost.
- Copies of bit 0 of src1 are supplied to the vacated bits on the left.

The result is placed into word element i of VSR[VRT+32].

Special Registers Altered:

None

Vector Shift Right Algebraic Doubleword VX-form

vsrad VRT,VRA,VRB

4	VRT	VRA	VRB	964
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 1
   src1 ← VSR[VRA+32].dword[i]
src2 ← VSR[VRB+32].dword[i].bit[58:63]
   VSR[VRT+32].dword[i] ← CHOP64(EXTS(src1) >> src2)
end
```

For each integer value i from 0 to 1, do the following.

Let src1 be the contents of doubleword element i of VSR[VRA+32].
Let src2 be the contents of doubleword element i of VSR[VRB+32].

src1 is shifted right by the number of bits specified in the low-order 6 bits of src2.

- Bits shifted out the least-significant bit are lost.
- Copies of bit 0 of src1 are supplied to the vacated bits on the left.

The result is placed into doubleword element i of VSR[VRT+32].

Special Registers Altered:

None

Register Data Layout for vsraw

src1	VSR[VRA+32].word[0]	VSR[VRA+32].word[1]	VSR[VRA+32].word[2]	VSR[VRA+32].word[3]
src2	VSR[VRB+32].word[0]	VSR[VRB+32].word[1]	VSR[VRB+32].word[2]	VSR[VRB+32].word[3]
result	VSR[VRT+32].word[0]	VSR[VRT+32].word[1]	VSR[VRT+32].word[2]	VSR[VRT+32].word[3]
	0	32	64	96 127

Register Data Layout for vsrad

src1	VSR[VRA+32].dword[0]	VSR[VRA+32].dword[1]
src2	VSR[VRB+32].dword[0]	VSR[VRB+32].dword[1]
result	VSR[VRT+32].dword[0]	VSR[VRT+32].dword[1]
	0	64 127

Vector Shift Right Algebraic Quadword VX-form

vsraq VRT,VRA,VRB

4	VRT	VRA	VRB	773
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

src1 ← VSR[VRA+32]
src2 ← VSR[VRB+32].bit[57:63]
VSR[VRT+32] ← CHOP128(EXTS(src1) >> src2)
```

Let `src1` be the contents of `VSR[VRA+32]`.
Let `src2` be the contents of `VSR[VRB+32]`.

`src1` is shifted right by the number of bits specified in the low-order 7 bits of `src2`.

- Bits shifted out the least-significant bit are lost.
- Copies of bit 0 of `src1` are supplied to the vacated bits on the left.

The result is placed into `VSR[VRT+32]`.

Special Registers Altered:

None

Register Data Layout for vsraq

6.10 Vector Floating-Point Instruction Set

6.10.1 Vector Floating-Point Arithmetic Instructions

Vector Add Floating-Point VX-form

vaddfp VRT,VRA,VRB

4	VRT	VRA	VRB	10
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
   src1 ← VSR[VRA+32].word[i]
   src2 ← VSR[VRB+32].word[i]
   VSR[VRT+32].word[i] ← bfp32_ADD(src1,src2)
end
```

For each integer value `i` from 0 to 3, do the following.

Let `src1` be the single-precision floating-point value in word element `i` of `VSR[VRA+32]`.
Let `src2` be the single-precision floating-point value in word element `i` of `VSR[VRB+32]`.

`src1` is added to `src2`.

The intermediate result is rounded to the nearest single-precision floating-point number and placed into word element `i` of `VSR[VRT+32]`.

Special Registers Altered:

None

Vector Subtract Floating-Point VX-form

vsubfp VRT,VRA,VRB

4	VRT	VRA	VRB	74
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
   src1 ← VSR[VRA+32].word[i]
   src2 ← VSR[VRB+32].word[i]
   VSR[VRT+32].word[i] ← bfp32_SUBTRACT(src1,src2)
end
```

For each integer value `i` from 0 to 3, do the following.

Let `src1` be the single-precision floating-point value in word element `i` of `VSR[VRA+32]`.
Let `src2` be the single-precision floating-point value in word element `i` of `VSR[VRB+32]`.

`src2` is subtracted from `src1`.

The intermediate result is rounded to the nearest single-precision floating-point number and placed into word element `i` of `VSR[VRT+32]`.

Special Registers Altered:

None

Register Data Layout for vaddfp & vsubfp

src1	VSR[VRA+32].word[0]	VSR[VRA+32].word[1]	VSR[VRA+32].word[2]	VSR[VRA+32].word[3]
src2	VSR[VRB+32].word[0]	VSR[VRB+32].word[1]	VSR[VRB+32].word[2]	VSR[VRB+32].word[3]
result	VSR[VRT+32].word[0]	VSR[VRT+32].word[1]	VSR[VRT+32].word[2]	VSR[VRT+32].word[3]
	0	32	64	96 127

Vector Multiply-Add Floating-Point VA-form

vmaddfp VRT,VRA,VRC,VRB

4	VRT	VRA	VRB	VRC	46
0	6	11	16	21	26 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
   src1 ← VSR[VRA+32].word[i]
   src2 ← VSR[VRB+32].word[i]
   src3 ← VSR[VRC+32].word[i]
   result ← bfp32_MULTIPLY_ADD(src1,src3,src2)
   VSR[VRT+32].word[i] ← result
end
```

For each integer value i from 0 to 3, do the following.

Let src1 be the single-precision floating-point value in word element i of VSR[VRA+32].
Let src2 be the single-precision floating-point value in word element i of VSR[VRB+32].
Let src3 be the single-precision floating-point value in word element i of VSR[VRC+32].

src1 is multiplied by src3.

src2 is added to the infinitely-precise product.

The intermediate result is rounded to the nearest single-precision floating-point number and placed into word element i of VSR[VRT+32].

Special Registers Altered:

None

Programming Note

To use a multiply-add to perform an IEEE or Java compliant multiply, the addend must be −0.0. This is necessary to insure that the sign of a zero result will be correct when the product is −0.0 (+0.0 + −0.0 ≥ +0.0, and −0.0 + -0.0 ≥ −0.0). When the sign of a resulting 0.0 is not important, then +0.0 can be used as an addend which may, in some cases, avoid the need for a second register to hold a −0.0 in addition to the integer floating-point +0.0 that may already be available.

Vector Negative Multiply-Subtract Floating-Point VA-form

vnmsubfp VRT,VRA,VRC,VRB

4	VRT	VRA	VRB	VRC	47
0	6	11	16	21	26 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
   src1 ← VSR[VRA+32].word[i]
   src2 ← VSR[VRB+32].word[i]
   src3 ← VSR[VRC+32].word[i]
   result ←
      bfp32_NEGATIVE_MULTIPLY_SUBTRACT(src1,src3,src2)
   VSR[VRT+32].word[i] ← result
end
```

For each integer value i from 0 to 3, do the following.

Let src1 be the single-precision floating-point value in word element i of VSR[VRA+32].
Let src2 be the single-precision floating-point value in word element i of VSR[VRB+32].
Let src3 be the single-precision floating-point value in word element i of VSR[VRC+32].

src1 is multiplied by src3.

src2 is subtracted from the infinitely-precise product.

The intermediate result is rounded to the nearest single-precision floating-point number, then negated and placed into word element i of VSR[VRT+32].

Special Registers Altered:

None

Register Data Layout for vmaddfp & vnmsubfp

src1	VSR[VRA+32].word[0]	VSR[VRA+32].word[1]	VSR[VRA+32].word[2]	VSR[VRA+32].word[3]
src2	VSR[VRB+32].word[0]	VSR[VRB+32].word[1]	VSR[VRB+32].word[2]	VSR[VRB+32].word[3]
src3	VSR[VRC+32].word[0]	VSR[VRC+32].word[1]	VSR[VRC+32].word[2]	VSR[VRC+32].word[3]
result	VSR[VRT+32].word[0]	VSR[VRT+32].word[1]	VSR[VRT+32].word[2]	VSR[VRT+32].word[3]
	0	32	64	96 127

6.10.2 Vector Floating-Point Maximum/Minimum Instructions

Vector Maximum Floating-Point VX-form

vmaxfp VRT,VRA,VRB

4	VRT	VRA	VRB	1034
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
   src1 ← VSR[VRA+32].word[i]
   src2 ← VSR[VRB+32].word[i]
   VSR[VRT+32].word[i] ← bfp32_MAXIMUM(src1,src2)
end
```

For each integer value `i` from 0 to 3, do the following.

Let `src1` be the single-precision floating-point value in word element `i` of `VSR[VRA+32]`.
Let `src2` be the single-precision floating-point value in word element `i` of `VSR[VRB+32]`.

`src1` is compared to `src2`.

The larger of the two values is placed into word element `i` of `VSR[VRT+32]`.

The maximum of +0.0 and −0.0 is +0.0. The maximum of any value and a NaN is a QNaN.

Special Registers Altered:

None

Vector Minimum Floating-Point VX-form

vminfp VRT,VRA,VRB

4	VRT	VRA	VRB	1098
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
   src1 ← VSR[VRA+32].word[i]
   src2 ← VSR[VRB+32].word[i]
   VSR[VRT+32].word[i] ← bfp32_MINIMUM(src1,src2)
end
```

For each integer value `i` from 0 to 3, do the following.

Let `src1` be the single-precision floating-point value in word element `i` of `VSR[VRA+32]`.
Let `src2` be the single-precision floating-point value in word element `i` of `VSR[VRB+32]`.

`src1` is compared to `src2`.

The smaller of the two values is placed into word element `i` of `VSR[VRT+32]`.

The minimum of +0.0 and −0.0 is −0.0. The minimum of any value and a NaN is a QNaN.

Special Registers Altered:

None

Register Data Layout for vmaxfp & vminfp

src1	VSR[VRA+32].word[0]	VSR[VRA+32].word[1]	VSR[VRA+32].word[2]	VSR[VRA+32].word[3]
src2	VSR[VRB+32].word[0]	VSR[VRB+32].word[1]	VSR[VRB+32].word[2]	VSR[VRB+32].word[3]

result	VSR[VRT+32].word[0]	VSR[VRT+32].word[1]	VSR[VRT+32].word[2]	VSR[VRT+32].word[3]
	0	32	64	96 127

6.10.3 Vector Floating-Point Rounding and Conversion Instructions

6.10.3.1 Vector Floating-Point Conversion Instructions

Vector Convert with round to zero from floating-point To Signed Word format Saturate VX-form

vctsxs VRT,VRB,UIM

4	VRT	UIM	VRB	970
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
   src ← VSR[VRB+32].word[i]
   VSR[VRT+32].word[i] ←
      si32_CONVERT_FROM_BFP32(src, UIM)
end
```

For each integer value `i` from 0 to 3, do the following.

Let `src` be the signed floating-point value in word element `i` of `VSR[VRB+32]`.

`src` is multiplied by 2^{UIM}. The product is converted to a 32-bit signed fixed-point integer using the rounding mode Round toward Zero.

- If the intermediate result is greater than $2^{31}-1$, the result saturates to $2^{31}-1$.
- If the intermediate result is less than -2^{31}, the result saturates to -2^{31}.

The result is placed into word element `i` of `VSR[VRT+32]`.

Special Registers Altered:

Register	*Field(s)*
VSCR	SAT

Extended Mnemonics:

Extended Mnemonics for *Vector Convert to Signed Fixed-Point Word Saturate*:

Extended mnemonic:	Equivalent to:
`vcfpsxws VRT,VRB,UIM`	`vctsxs VRT,VRB,UIM`

Vector Convert with round to zero from floating-point To Unsigned Word format Saturate VX-form

vctuxs VRT,VRB,UIM

4	VRT	UIM	VRB	906
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
   src ← VSR[VRB+32].word[i]
   VSR[VRT+32].word[i] ←
      ui32_CONVERT_FROM_BFP32(src, UIM)
end
```

For each integer value `i` from 0 to 3, do the following.

Let `src` be the signed floating-point value in word element `i` of `VSR[VRB+32]`.

`src` is multiplied by 2^{UIM}. The product is converted to a 32-bit unsigned fixed-point integer using the rounding mode Round toward Zero.

- If the intermediate result is greater than $2^{32}-1$, the result saturates to $2^{32}-1$.

The result is placed into word element `i` of `VSR[VRT+32]`.

Special Registers Altered:

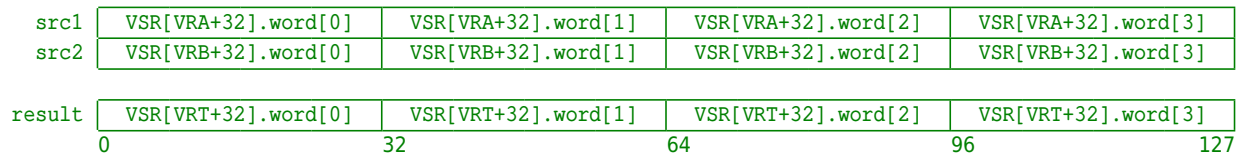
Register	*Field(s)*
VSCR	SAT

Extended Mnemonics:

Extended Mnemonics for *Vector Convert to Unsigned Fixed-Point Word Saturate*:

Extended mnemonic:	Equivalent to:
`vcfpuxws VRT,VRB,UIM`	`vctuxs VRT,VRB,UIM`

Register Data Layout for vctsxs & vctuxs



Vector Convert with round to nearest From Signed Word to floating-point format VX-form

vcfsx VRT,VRB,UIM

4	VRT	UIM	VRB	842
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
   src ← VSR[VRB+32].word[i]
   VSR[VRT+32].word[i] ←
      bfp32_CONVERT_FROM_SI32(src,UIM)
end
```

For each integer value `i` from 0 to 3, do the following.

Let `src` be the signed fixed-point value in word element `i` of `VSR[VRB+32]`.

`src` is converted to the nearest single-precision floating-point value. Each result is divided by 2^{UIM} and placed into word element `i` of `VSR[VRT+32]`.

Special Registers Altered:

None

Extended Mnemonics:

Extended Mnemonics for *Vector Convert from Signed Fixed-Point Word*:

Extended mnemonic:	Equivalent to:
`vcsxwfp VRT,VRB,UIM`	`vcfsx VRT,VRB,UIM`

Vector Convert with round to nearest From Unsigned Word to floating-point format VX-form

vcfux VRT,VRB,UIM

4	VRT	UIM	VRB	778
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
   src ← VSR[VRB+32].word[i]
   VSR[VRT+32].word[i] ←
      bfp32_CONVERT_FROM_UI32(src,UIM)
end
```

For each integer value `i` from 0 to 3, do the following.

Let `src` be the unsigned fixed-point value in word element `i` of `VSR[VRB+32]`.

`src` is converted to the nearest single-precision floating-point value. The result is divided by 2^{UIM} and placed into word element `i` of `VSR[VRT+32]`.

Special Registers Altered:

None

Extended Mnemonics:

Extended Mnemonics for *Vector Convert from Unsigned Fixed-Point Word*:

Extended mnemonic:	Equivalent to:
`vcuxwfp VRT,VRB,UIM`	`vcfux VRT,VRB,UIM`

Register Data Layout for vcfsx & vcfux

src1	VSR[VRA+32].word[0]	VSR[VRA+32].word[1]	VSR[VRA+32].word[2]	VSR[VRA+32].word[3]
src2	VSR[VRB+32].word[0]	VSR[VRB+32].word[1]	VSR[VRB+32].word[2]	VSR[VRB+32].word[3]

result	VSR[VRT+32].word[0]	VSR[VRT+32].word[1]	VSR[VRT+32].word[2]	VSR[VRT+32].word[3]
	0	32	64	96 127

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.

6.10.3.2 Vector Floating-Point Round to Integral Instructions

Vector Round to Floating-Point Integer toward –Infinity VX-form

vrfim VRT,VRB

4	VRT	///	VRB	714
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
   src ← VSR[VRB+32].word[i]
   VSR[VRT+32].word[i] ←
      bfp32_ROUND_TO_INTEGER_FLOOR(src)
end
```

For each integer value i from 0 to 3, do the following.

Let src be the single-precision floating-point value in element i of VSR[VRB+32].

src is rounded to a single-precision floating-point integer using the rounding mode Round toward -Infinity.

The result is placed into the corresponding word element i of VSR[VRT+32].

Special Registers Altered:

None

Vector Round to Floating-Point Integer Nearest VX-form

vrfin VRT,VRB

4	VRT	///	VRB	522
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
   src ← VSR[VRB+32].word[i]
   VSR[VRT+32].word[i] ←
      bfp32_ROUND_TO_INTEGER_NEAR(src)
end
```

For each integer value i from 0 to 3, do the following.

Let src be the single-precision floating-point value in element i of VSR[VRB+32].

src is rounded to a single-precision floating-point integer using the rounding mode Round to Nearest.

The result is placed into the corresponding word element i of VSR[VRT+32].

Special Registers Altered:

None

Register Data Layout for vrfim & vrfin

src1	VSR[VRA+32].word[0]	VSR[VRA+32].word[1]	VSR[VRA+32].word[2]	VSR[VRA+32].word[3]
src2	VSR[VRB+32].word[0]	VSR[VRB+32].word[1]	VSR[VRB+32].word[2]	VSR[VRB+32].word[3]

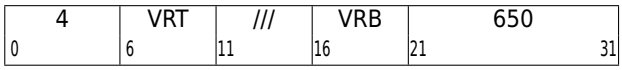
result	VSR[VRT+32].word[0]	VSR[VRT+32].word[1]	VSR[VRT+32].word[2]	VSR[VRT+32].word[3]
	0	32	64	96 127

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.

Vector Round to Floating-Point Integer toward +Infinity VX-form

vrfip VRT,VRB




```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
   src ← VSR[VRB+32].word[i]
   VSR[VRT+32].word[i] ←
      bfp32_ROUND_TO_INTEGER_CEIL(src)
end
```

For each integer value `i` from 0 to 3, do the following.

Let `src` be the single-precision floating-point value in element `i` of `VSR[VRB+32]`.

`src` is rounded to a single-precision floating-point integer using the rounding mode Round toward +Infinity.

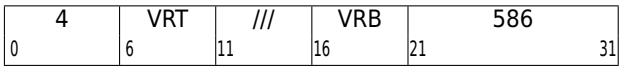
The result is placed into the corresponding word element `i` of `VSR[VRT+32]`.

Special Registers Altered:

None

Vector Round to Floating-Point Integer toward Zero VX-form

vrfiz VRT,VRB




```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
   src ← VSR[VRB+32].word[i]
   VSR[VRT+32].word[i] ←
      bfp32_ROUND_TO_INTEGER_TRUNC(src)
end
```

For each integer value `i` from 0 to 3, do the following.

Let `src` be the single-precision floating-point value in element `i` of `VSR[VRB+32]`.

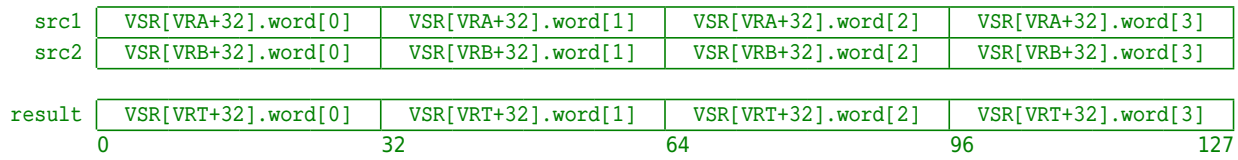
`src` is rounded to a single-precision floating-point integer using the rounding mode Round toward Zero.

The result is placed into the corresponding word element `i` of `VSR[VRT+32]`.

Special Registers Altered:

None

Register Data Layout for vrfip & vrfiz



6.10.4 Vector Floating-Point Compare Instructions

The *Vector Floating-Point Compare* instructions compare two VSRs word element by word element, interpreting the elements as single-precision floating-point numbers. With the exception of the *Vector Compare Bounds Floating-Point* instruction, they set the target VSR, and CR Field 6 if Rc=1, in the same manner as do the *Vector Integer Compare* instructions; see Section 6.9.3.

The *Vector Compare Bounds Floating-Point* instruction sets the target VSR, and CR Field 6 if Rc=1, to indicate whether the elements in VSR[VRA+32] are within the bounds specified by the corresponding element in VSR[VRB+32], as explained in the instruction description. A single-precision floating-point value x is said to be "within the bounds" specified by a single-precision floating-point value y if $-y \leq x \leq y$.

Vector Compare Bounds Floating-Point VC-form

vcmpbfp	VRT,VRA,VRB	(Rc=0)
vcmpbfp.	VRT,VRA,VRB	(Rc=1)

4	VRT	VRA	VRB	Rc	966
0	6	11	16	21	22 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
   src1 ← VSR[VRA+32].word[i]
   src2 ← VSR[VRB+32].word[i]
   le ← bool_COMPARE_LE_BFP32(src1,src2)
   ge ← bool_COMPARE_GE_BFP32(src1,src2)
   VSR[VRT+32].word[i] ← ¬le || ¬ge || 30 0
end
if Rc=1 then do
   ib ← (VSR[VRT+32]=0)
   CR6 ← 0b00 || ib || 0b0
end
```

For each integer value i from 0 to 3, do the following.

Let src1 be the single-precision floating-point value in element i of VSR[VRA+32].
Let src2 be the single-precision floating-point value in element i of VSR[VRB+32].

src1 is compared to src2.

A 2-bit value is formed that indicates whether src1 is within the bounds specified by src2, as follows.

- Bit 0 of the 2-bit value is set to 0 if src1 is less than or equal to src2, and is set to 1 otherwise.
- Bit 1 of the 2-bit value is set to 0 if src1 is greater than or equal to the negation of src2, and is set to 1 otherwise.

The 2-bit value is placed into the high-order two bits of word element i of VSR[VRT+32] and the remaining bits of element i are set to 0.

If Rc=1, CR field 6 is set as follows.

Bit	Description
0	Set to 0
1	Set to 0
2	Set to indicate whether all four elements in VSR[VRA+32] are within the bounds specified by the corresponding element in VSR[VRB+32], otherwise set to 0.
3	Set to 0

Special Registers Altered:

Register	*Field(s)*	
CR	CR6	(if Rc=1)

Programming Note

Each single-precision floating-point value in VSR[VRB+32] should be non-negative; if it is negative, the corresponding element in VSR[VRA+32] will necessarily be out of bounds.

One exception to this is when the value of an element in VSR[VRB+32] is −0.0 and the value of the corresponding element in VSR[VRA+32] is either +0.0 or −0.0. +0.0 and −0.0 compare equal to −0.0.

Register Data Layout for vcmpbfp[.]

src1	VSR[VRA+32].word[0]	VSR[VRA+32].word[1]	VSR[VRA+32].word[2]	VSR[VRA+32].word[3]
src2	VSR[VRB+32].word[0]	VSR[VRB+32].word[1]	VSR[VRB+32].word[2]	VSR[VRB+32].word[3]

result	VSR[VRT+32].word[0]	VSR[VRT+32].word[1]	VSR[VRT+32].word[2]	VSR[VRT+32].word[3]
	0	32	64	96 127

Vector Compare Equal Floating-Point VC-form

vcmpeqfp	VRT,VRA,VRB	(Rc=0)
vcmpeqfp.	VRT,VRA,VRB	(Rc=1)

4	VRT	VRA	VRB	Rc	198
0	6	11	16	21	22 31

```
if MSR.VEC=0 then Vector_Unavailable()

all_true  ← 1
all_false ← 1
do i = 0 to 3
   src1 ← VSR[VRA+32].word[i]
   src2 ← VSR[VRB+32].word[i]
   if bool_COMPARE_EQ_BFP32(src1,src2)=1 then
      VSR[VRT+32].word[i] ← 0xFFFF_FFFF
      all_false ← 0
   else
      VSR[VRT+32].word[i] ← 0x0000_0000
      all_true ← 0
end
if Rc=1 then
   CR.field[6] ← all_true || 0b0 || all_false || 0b0
```

For each integer value `i` from 0 to 3, do the following.

Let `src1` be the single-precision floating-point value in element `i` of `VSR[VRA+32]`.
Let `src2` be the single-precision floating-point value in element `i` of `VSR[VRB+32]`.

`src1` is compared to `src2`.

The contents of word element `i` of `VSR[VRT+32]` are set to all 1s if `src1` is equal to `src2`, and are set to all 0s otherwise.

If `src1` or `src2` is a NaN, the contents of word element `i` of `VSR[VRT+32]` are set to all 0s, indicating "not equal to". If `src1` and `src2` are both infinity with the same sign, the contents of word element `i` of `VSR[VRT+32]` are set to all 1s, indicating "equal to".

If `Rc=1`, CR field 6 is set to indicate whether all vector elements compared true and whether all vector elements compared false.

Special Registers Altered:

Register	*Field(s)*	
CR	CR6	(if Rc=1)

Vector Compare Greater Than or Equal Floating-Point VC-form

vcmpgefp	VRT,VRA,VRB	(Rc=0)
vcmpgefp.	VRT,VRA,VRB	(Rc=1)

4	VRT	VRA	VRB	Rc	454
0	6	11	16	21	22 31

```
if MSR.VEC=0 then Vector_Unavailable()
all_true  ← 1
all_false ← 1
do i = 0 to 3
   src1 ← VSR[VRA+32].word[i]
   src2 ← VSR[VRB+32].word[i]
   if bool_COMPARE_GE_BFP32(src1,src2)=1 then
      VSR[VRT+32].word[i] ← 0xFFFF_FFFF
      all_false ← 0
   else
      VSR[VRT+32].word[i] ← 0x0000_0000
      all_true ← 0
end
if Rc=1 then
   CR.field[6] ← all_true || 0b0 || all_false || 0b0
```

For each integer value `i` from 0 to 3, do the following.

Let `src1` be the single-precision floating-point value in element `i` of `VSR[VRA+32]`.
Let `src2` be the single-precision floating-point value in element `i` of `VSR[VRB+32]`.

`src1` is compared to `src2`.

The contents of word element `i` of `VSR[VRT+32]` are set to all 1s if `src1` is greater than or equal to `src2`, and are set to all 0s otherwise.

If `src1` or `src2` is a NaN, the contents of word element `i` of `VSR[VRT+32]` are set to all 0s, indicating "not greater than or equal to". If `src1` and `src2` are both infinity with the same sign, the contents of word element `i` of `VSR[VRT+32]` are set to all 1s, indicating "greater than or equal to".

If `Rc=1`, CR field 6 is set to indicate whether all vector elements compared true and whether all vector elements compared false.

Special Registers Altered:

Register	*Field(s)*	
CR	CR6	(if Rc=1)

Register Data Layout for vcmpeqfp[.] & vcmpgefp[.]

src1	VSR[VRA+32].word[0]	VSR[VRA+32].word[1]	VSR[VRA+32].word[2]	VSR[VRA+32].word[3]
src2	VSR[VRB+32].word[0]	VSR[VRB+32].word[1]	VSR[VRB+32].word[2]	VSR[VRB+32].word[3]

result	VSR[VRT+32].word[0]	VSR[VRT+32].word[1]	VSR[VRT+32].word[2]	VSR[VRT+32].word[3]
	0	32	64	96 127

Vector Compare Greater Than Floating-Point VC-form

vcmpgtfp	VRT,VRA,VRB	(Rc=0)
vcmpgtfp.	VRT,VRA,VRB	(Rc=1)

4	VRT	VRA	VRB	Rc	710
0	6	11	16	21	22 31

```
if MSR.VEC=0 then Vector_Unavailable()

all_true  ← 1
all_false ← 1

do i = 0 to 3
   src1 ← VSR[VRA+32].word[i]
   src2 ← VSR[VRB+32].word[i]
   if bool_COMPARE_GT_BFP32(src1,src2)=1 then
      VSR[VRT+32].word[i] ← 0xFFFF_FFFF
      all_false ← 0
   else
      all_true ← 0
      VSR[VRT+32].word[i] ← 0x0000_0000
end
if Rc=1 then
   CR.field[6] ← all_true || 0b0 || all_false || 0b0
```

For each integer value `i` from 0 to 3, do the following.

Let `src1` be the single-precision floating-point value in element `i` of `VSR[VRA+32]`.
Let `src2` be the single-precision floating-point value in element `i` of `VSR[VRB+32]`.

`src1` is compared to `src2`.

The contents of word element `i` of `VSR[VRT+32]` are set to all 1s if `src1` is greater than `src2`, and are set to all 0s otherwise.

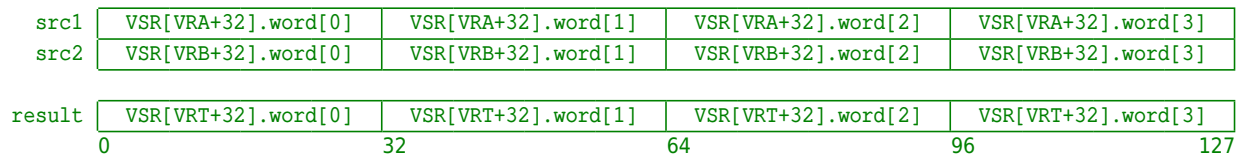
If `src1` or `src2` is a NaN, the contents of word element `i` of `VSR[VRT+32]` are set to all 0s, indicating "not greater than". If `src1` and `src2` are both infinity with the same sign, the contents of word element `i` of `VSR[VRT+32]` are set to all 0s, indicating "not greater than".

If `Rc=1`, CR field 6 is set to indicate whether all vector elements compared true and whether all vector elements compared false.

Special Registers Altered:

Register	*Field(s)*	
CR	CR6	(if Rc=1)

Register Data Layout for vcmpgtfp[.]



6.10.5 Vector Floating-Point Estimate Instructions

Vector 2 Raised to the Exponent Estimate Floating-Point VX-form

vexptefp VRT,VRB

4	VRT	///	VRB	394
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
   src ← VSR[VRB+32].word[i]
   VSR[VRT+32].word[i] ← bfp32_POWER2_ESTIMATE(src)
end
```

For each integer value `i` from 0 to 3, do the following.

The single-precision floating-point estimate of 2 raised to the power of single-precision floating-point element `i` of `VSR[VRB+32]` is placed into word element `i` of `VSR[VRT+32]`.

Let `x` be any single-precision floating-point input value. Unless $x < -146$ or the single-precision floating-point result of computing 2 raised to the power `x` would be a zero, an infinity, or a QNaN, the estimate has a relative error in precision no greater than one part in 16. The most significant 12 bits of the estimate's significand are monotonic. An integral input value returns an integral value when the result is representable.

The result for various special cases of the source value is given below.

Value	Result
−Infinity	+0
−0	+1
+0	+1
+Infinity	+Infinity
NaN	QNaN

Special Registers Altered:

None

Register Data Layout for vexptefp

src1	VSR[VRA+32].word[0]	VSR[VRA+32].word[1]	VSR[VRA+32].word[2]	VSR[VRA+32].word[3]
src2	VSR[VRB+32].word[0]	VSR[VRB+32].word[1]	VSR[VRB+32].word[2]	VSR[VRB+32].word[3]
result	VSR[VRT+32].word[0]	VSR[VRT+32].word[1]	VSR[VRT+32].word[2]	VSR[VRT+32].word[3]
	0	32	64	96 127

Vector Log Base 2 Estimate Floating-Point VX-form

vlogefp VRT,VRB

4	VRT	///	VRB	458
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
   src ← VSR[VRB+32].word[i]
   VSR[VRT+32].word[i] ← bfp32_LOG_BASE2_ESTIMATE(src)
end
```

For each integer value `i` from 0 to 3, do the following.

> The single-precision floating-point estimate of the base 2 logarithm of single-precision floating-point element `i` of `VSR[VRB+32]` is placed into the corresponding word element `i` of `VSR[VRT+32]`.

Let `x` be any single-precision floating-point input value. Unless $|x-1|$ is less than or equal to 0.125 or the single-precision floating-point result of computing the base 2 logarithm of `x` would be an infinity or a QNaN, the estimate has an absolute error in precision (absolute value of the difference between the estimate and the infinitely precise value) no greater than 2^{-5}. Under the same conditions, the estimate has a relative error in precision no greater than one part in 8.

The most significant 12 bits of the estimate's significand are monotonic. The estimate is exact if $x=2^y$, where y is an integer between -149 and +127 inclusive. Otherwise the value placed into the element of `VSR[VRT+32]` may vary between implementations, and between different executions on the same implementation.

The result for various special cases of the source value is given below.

Value	Result
−Infinity	QNaN
< 0	QNaN
−0	−Infinity
+0	−Infinity
+Infinity	+Infinity
NaN	QNaN

Special Registers Altered:

None

Register Data Layout for vlogefp

src1	VSR[VRA+32].word[0]	VSR[VRA+32].word[1]	VSR[VRA+32].word[2]	VSR[VRA+32].word[3]
src2	VSR[VRB+32].word[0]	VSR[VRB+32].word[1]	VSR[VRB+32].word[2]	VSR[VRB+32].word[3]

result	VSR[VRT+32].word[0]	VSR[VRT+32].word[1]	VSR[VRT+32].word[2]	VSR[VRT+32].word[3]
	0	32	64	96 127

Vector Reciprocal Estimate Floating-Point VX-form

vrefp VRT,VRB

4	VRT	///	VRB	266
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
   src ← VSR[VRB+32].word[i]
   VSR[VRT+32].word[i] ←
      bfp32_RECIPROCAL_ESTIMATE(src)
end
```

For each integer value `i` from `0` to `3`, do the following.

> The single-precision floating-point estimate of the reciprocal of single-precision floating-point element `i` of `VSR[VRB+32]` is placed into word element `i` of `VSR[VRT+32]`.

Unless the single-precision floating-point result of computing the reciprocal of a value would be a zero, an infinity, or a QNaN, the estimate has a relative error in precision no greater than one part in 4096.

Note that results may vary between implementations, and between different executions on the same implementation.

The result for various special cases of the source value is given below.

Value	Result
−Infinity	−0
−0	−Infinity
+0	+Infinity
+Infinity	+0
NaN	QNaN

Special Registers Altered:

None

Vector Reciprocal Square Root Estimate Floating-Point VX-form

vrsqrtefp VRT,VRB

4	VRT	///	VRB	330
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
   src ← VSR[VRB+32].word[i]
   VSR[VRT+32].word[i] ←
      bfp32_RECIPROCAL_SQRT_ESTIMATE(src)
end
```

For each integer value `i` from `0` to `3`, do the following.

> The single-precision floating-point estimate of the reciprocal of the square root of single-precision floating-point element `i` of `VSR[VRB+32]` is placed into word element `i` of `VSR[VRT+32]`.

Let `x` be any single-precision floating-point value. Unless the single-precision floating-point result of computing the reciprocal of the square root of `x` would be a zero, an infinity, or a QNaN, the estimate has a relative error in precision no greater than one part in 4096.

Note that results may vary between implementations, and between different executions on the same implementation.

The result for various special cases of the source value is given below.

Value	Result
−Infinity	QNaN
< 0	QNaN
−0	−Infinity
+0	+Infinity
+Infinity	+0
NaN	QNaN

Special Registers Altered:

None

Register Data Layout for vrefp & vrsqrtefp

src1	VSR[VRA+32].word[0]	VSR[VRA+32].word[1]	VSR[VRA+32].word[2]	VSR[VRA+32].word[3]
src2	VSR[VRB+32].word[0]	VSR[VRB+32].word[1]	VSR[VRB+32].word[2]	VSR[VRB+32].word[3]

result	VSR[VRT+32].word[0]	VSR[VRT+32].word[1]	VSR[VRT+32].word[2]	VSR[VRT+32].word[3]
	0	32	64	96 127

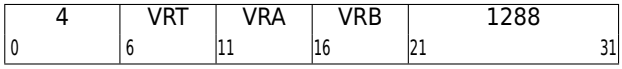
6.11 Vector Exclusive-OR-based Instructions

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

vcipher VRT,VRA,VRB




```
if MSR.VEC=0 then Vector_Unavailable()

State    ← VSR[VRA+32]
RoundKey ← VSR[VRB+32]
vtemp1   ← SubBytes(State)
vtemp2   ← ShiftRows(vtemp1)
vtemp3   ← MixColumns(vtemp2)
VSR[VRT+32]  ← vtemp3 ⊕ RoundKey
```

Let `State` be the contents of `VSR[VRA+32]`, representing the intermediate state array during AES cipher operation.

Let `RoundKey` be the contents of `VSR[VRB+32]`, representing the round key.

One round of an AES cipher operation is performed on the intermediate `State` array, sequentially applying the transforms, `SubBytes()`, `ShiftRows()`, `MixColumns()`, and `AddRoundKey()`, as defined in FIPS-197.

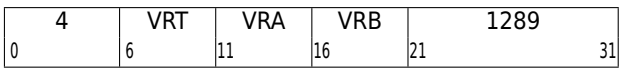
The result is placed into `VSR[VRT+32]`, representing the new intermediate state of the cipher operation.

Special Registers Altered:

None

Vector AES Cipher Last VX-form

vcipherlast VRT,VRA,VRB




```
if MSR.VEC=0 then Vector_Unavailable()

State    ← VSR[VRA+32]
RoundKey ← VSR[VRB+32]
vtemp1   ← SubBytes(State)
vtemp2   ← ShiftRows(vtemp1)
VSR[VRT+32]  ← vtemp2 ⊕ RoundKey
```

Let `State` be the contents of `VSR[VRA+32]`, representing the intermediate state array during AES cipher operation.

Let `RoundKey` be the contents of `VSR[VRB+32]`, representing the round key.

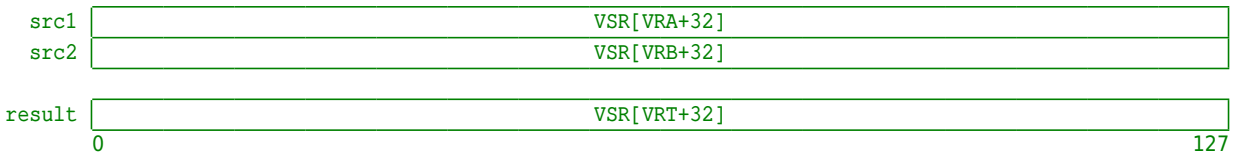
The final round in an AES cipher operation is performed on the intermediate `State` array, sequentially applying the transforms, `SubBytes()`, `ShiftRows()`, `AddRoundKey()`, as defined in FIPS-197.

The result is placed into `VSR[VRT+32]`, representing the final state of the cipher operation.

Special Registers Altered:

None

Register Data Layout for vcipher & vcipherlast



Vector AES Inverse Cipher VX-form

vncipher VRT,VRA,VRB

4	VRT	VRA	VRB	1352
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

State    ← VSR[VRA+32]
RoundKey ← VSR[VRB+32]
vtemp1   ← InvShiftRows(State)
vtemp2   ← InvSubBytes(vtemp1)
vtemp3   ← vtemp2 ⊕ RoundKey
VSR[VRT+32]  ← InvMixColumns(vtemp3)
```

Let State be the contents of VSR[VRA+32], representing the intermediate state array during AES inverse cipher operation.

Let RoundKey be the contents of VSR[VRB+32], representing the round key.

One round of an AES inverse cipher operation is performed on the intermediate State array, sequentially applying the transforms, InvShiftRows(), InvSubBytes(), AddRoundKey(), and InvMixColumns(), as defined in FIPS-197.

The result is placed into VSR[VRT+32], representing the new intermediate state of the inverse cipher operation.

Special Registers Altered:

None

Vector AES Inverse Cipher Last VX-form

vncipherlast VRT,VRA,VRB

4	VRT	VRA	VRB	1353
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

State    ← VSR[VRA+32]
RoundKey ← VSR[VRB+32]
vtemp1   ← InvShiftRows(State)
vtemp2   ← InvSubBytes(vtemp1)
VSR[VRT+32]  ← vtemp2 ⊕ RoundKey
```

Let State be the contents of VSR[VRA+32], representing the intermediate state array during AES inverse cipher operation.

Let RoundKey be the contents of VSR[VRB+32], representing the round key.

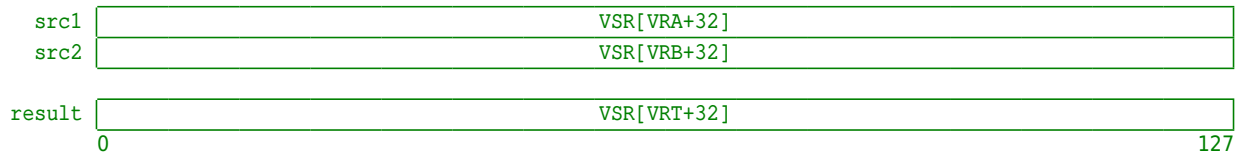
The final round in an AES inverse cipher operation is performed on the intermediate State array, sequentially applying the transforms, InvShiftRows(), InvSubBytes(), and AddRoundKey(), as defined in FIPS-197.

The result is placed into VSR[VRT+32], representing the final state of the inverse cipher operation.

Special Registers Altered:

None

Register Data Layout for vncipher & vncipherlast



Vector AES SubBytes VX-form

vsbox VRT,VRA

4	VRT	VRA	///	1480
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

State   ← VSR[VRA+32]
VSR[VRT+32] ← SubBytes(State)
```

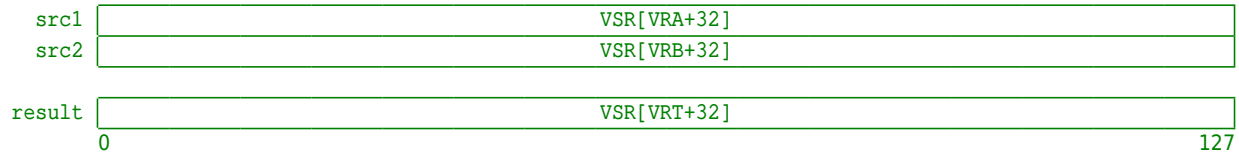
Let `State` be the contents of `VSR[VRA+32]`, representing the intermediate state array during AES cipher operation.

The result of applying the transform, `SubBytes()` on `State`, as defined in FIPS-197, is placed into `VSR[VRT+32]`.

Special Registers Altered:

None

Register Data Layout for vsbox



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

vshasigmad VRT,VRA,ST,SIX

4	VRT	VRA	ST	SIX	1730
0	6	11	16	17	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 1
   src ← VSR[VRA+32].dword[i]

   if ST=0 & SIX.bit[2×i]=0 then
      // SHA-512 σ0 function
      VSR[VRT+32].dword[i] ← (src >>>  1) ⊕
                  (src >>>  8) ⊕ (src >> 7)

   if ST=0 & SIX.bit[2×i]=1 then
      // SHA-512 σ1 function
      VSR[VRT+32].dword[i] ← (src >>> 19) ⊕
                  (src >>> 61) ⊕ (src >> 6)

   if ST=1 & SIX.bit[2×i]=0 then
      // SHA-512 Σ0 function
      VSR[VRT+32].dword[i] ← (src >>> 28) ⊕
                  (src >>> 34) ⊕ (src >>> 39)

   if ST=1 & SIX.bit[2×i]=1 then
      // SHA-512 Σ1 function
      VSR[VRT+32].dword[i] ← (src >>> 14) ⊕
                  (src >>> 18) ⊕ (src >>> 41)
end
```

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 VSR[VRA+32] and the result is placed into doubleword element i of VSR[VRT+32].

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 VSR[VRA+32] and the result is placed into doubleword element i of VSR[VRT+32].

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 VSR[VRA+32] and the result is placed into doubleword element i of VSR[VRT+32].

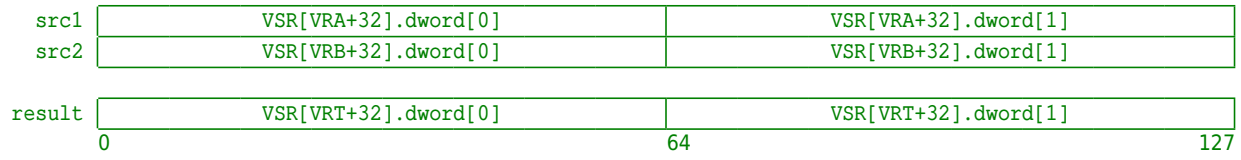
When ST=1 and bit 2×i of SIX is 1, a SHA-512 Σ1 function is performed on the contents of doubleword element i of VSR[VRA+32] and the result is placed into doubleword element i of VSR[VRT+32].

Bits 1 and 3 of SIX are reserved.

Special Registers Altered:

None

Register Data Layout for vshasigmad



Vector SHA-256 Sigma Word VX-form

vshasigmaw VRT,VRA,ST,SIX

4	VRT	VRA	ST	SIX	1666
0	6	11	16	17	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
   src ← VSR[VRA+32].word[i]

   if ST=0 & SIX.bit[i]=0 then
      // SHA-256 σ0 function
      VSR[VRT+32].word[i] ← (src >>>  7) ⊕
                 (src >>> 18) ⊕ (src >>  3)

   if ST=0 & SIX.bit[i]=1 then
      // SHA-256 σ1 function
      VSR[VRT+32].word[i] ← (src >>> 17) ⊕
                 (src >>> 19) ⊕ (src >> 10)

   if ST=1 & SIX.bit[i]=0 then
      // SHA-256 Σ0 function
      VSR[VRT+32].word[i] ← (src >>>  2) ⊕
                 (src >>> 13) ⊕ (src >>> 22)

   if ST=1 & SIX.bit[i]=1 then
      // SHA-256 Σ1 function
      VSR[VRT+32].word[i] ← (src >>>  6) ⊕
                 (src >>> 11) ⊕ (src >>> 25)
end
```

For each integer value `i` from 0 to 3, do the following.

When `ST=0` and bit `i` of `SIX` is `0`, a SHA-256 $\sigma 0$ function is performed on the contents of word element `i` of `VSR[VRA+32]` and the result is placed into word element `i` of `VSR[VRT+32]`.

When `ST=0` and bit `i` of `SIX` is `1`, a SHA-256 $\sigma 1$ function is performed on the contents of word element `i` of `VSR[VRA+32]` and the result is placed into word element `i` of `VSR[VRT+32]`.

When `ST=1` and bit `i` of `SIX` is `0`, a SHA-256 $\Sigma 0$ function is performed on the contents of word element `i` of `VSR[VRA+32]` and the result is placed into word element `i` of `VSR[VRT+32]`.

When `ST=1` and bit `i` of `SIX` is `1`, a SHA-256 $\Sigma 1$ function is performed on the contents of word element `i` of `VSR[VRA+32]` and the result is placed into word element `i` of `VSR[VRT+32]`.

Special Registers Altered:

None

Register Data Layout for vshasigmaw

src1	VSR[VRA+32].word[0]	VSR[VRA+32].word[1]	VSR[VRA+32].word[2]	VSR[VRA+32].word[3]
src2	VSR[VRB+32].word[0]	VSR[VRB+32].word[1]	VSR[VRB+32].word[2]	VSR[VRB+32].word[3]
result	VSR[VRT+32].word[0]	VSR[VRT+32].word[1]	VSR[VRT+32].word[2]	VSR[VRT+32].word[3]
	0	32	64	96 127

6.11.3 Vector Binary Polynomial Multiplication Instructions

This section describes a set of binary polynomial multiply-sum instructions. Corresponding elements are multiplied and the exclusive-OR of each even-odd pair of products sum, useful for a variety of finite field arithmetic operations.

Vector Polynomial Multiply-Sum Byte VX-form

vpmsumb VRT,VRA,VRB

4	VRT	VRA	VRB	1032
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 15
   prod[i].bit[0:14] ← 0
   srcA ← VSR[VRA+32].byte[i]
   srcB ← VSR[VRB+32].byte[i]
   do j = 0 to 7
      do k = 0 to j
         gbit ← srcA.bit[k] & srcB.bit[j-k]
         prod[i].bit[j] ← prod[i].bit[j] ⊕ gbit
      end
   end
   do j = 8 to 14
      do k = j-7 to 7
         gbit ← (srcA.bit[k] & srcB.bit[j-k])
         prod[i].bit[j] ← prod[i].bit[j] ⊕ gbit
      end
   end
end
do i = 0 to 7
VSR[VRT+32].hword[i] ←
   0b0 || (prod[2×i] ⊕ prod[2×i+1])
end
```

For each integer value `i` from 0 to 15, do the following.

Let `prod[i]` be the 15-bit result of a binary polynomial multiplication of the contents of byte element `i` of `VSR[VRA+32]` and the contents of byte element `i` of `VSR[VRB+32]`.

For each integer value `i` from 0 to 7, do the following.

The exclusive-OR of `prod[2×i]` and `prod[2×i+1]` is placed in bits 1:15 of halfword element `i` of `VSR[VRT+32]`. Bit 0 of halfword element `i` of `VSR[VRT+32]` is set to 0.

Special Registers Altered:

None

Vector Polynomial Multiply-Sum Halfword VX-form

vpmsumh VRT,VRA,VRB

4	VRT	VRA	VRB	1096
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 7
   prod.bit[0:30] ← 0
   srcA ← VSR[VRA+32].halfword[i]
   srcB ← VSR[VRB+32].halfword[i]
   do j = 0 to 15
      do k = 0 to j
         gbit ← srcA.bit[k] & srcB.bit[j-k]
         prod[i].bit[j] ← prod[i].bit[j] ⊕ gbit
      end
   end
   do j = 16 to 30
      do k = j-15 to 15
         gbit ← (srcA.bit[k] & srcB.bit[j-k])
         prod[i].bit[j] ← prod[i].bit[j] ⊕ gbit
      end
   end
end
VSR[VRT+32].word[0] ← 0b0 || (prod[0] ⊕ prod[1])
VSR[VRT+32].word[1] ← 0b0 || (prod[2] ⊕ prod[3])
VSR[VRT+32].word[2] ← 0b0 || (prod[4] ⊕ prod[5])
VSR[VRT+32].word[3] ← 0b0 || (prod[6] ⊕ prod[7])
```

For each integer value `i` from 0 to 7, do the following.

Let `prod[i]` be the 31-bit result of a binary polynomial multiplication of the contents of halfword element `i` of `VSR[VRA+32]` and the contents of halfword element `i` of `VSR[VRB+32]`.

For each integer value `i` from 0 to 3, do the following.

The exclusive-OR of `prod[2×i]` and `prod[2×i+1]` is placed in bits 1:31 of word element `i` of `VSR[VRT+32]`. Bit 0 of word element `i` of `VSR[VRT+32]` is set to 0.

Special Registers Altered:

None

Register Data Layout for vpmsumb

src1	.byte[0]	.byte[1]	.byte[2]	.byte[3]	.byte[4]	.byte[5]	.byte[6]	.byte[7]	.byte[8]	.byte[9]	.byte[10]	.byte[11]	.byte[12]	.byte[13]	.byte[14]	.byte[15]
src2	.byte[0]	.byte[1]	.byte[2]	.byte[3]	.byte[4]	.byte[5]	.byte[6]	.byte[7]	.byte[8]	.byte[9]	.byte[10]	.byte[11]	.byte[12]	.byte[13]	.byte[14]	.byte[15]

result	VSR[VRT+32].hword[0]	VSR[VRT+32].hword[1]	VSR[VRT+32].hword[2]	VSR[VRT+32].hword[3]	VSR[VRT+32].hword[4]	VSR[VRT+32].hword[5]	VSR[VRT+32].hword[6]	VSR[VRT+32].hword[7]
	0 8	16 24	32 40	48 56	64 72	80 88	96 104	112 120 127

Register Data Layout for vpmsumh

src1	VSR[VRA+32].hword[0]	VSR[VRA+32].hword[1]	VSR[VRA+32].hword[2]	VSR[VRA+32].hword[3]	VSR[VRA+32].hword[4]	VSR[VRA+32].hword[5]	VSR[VRA+32].hword[6]	VSR[VRA+32].hword[7]
src2	VSR[VRB+32].hword[0]	VSR[VRB+32].hword[1]	VSR[VRB+32].hword[2]	VSR[VRB+32].hword[3]	VSR[VRB+32].hword[4]	VSR[VRB+32].hword[5]	VSR[VRB+32].hword[6]	VSR[VRB+32].hword[7]

result	VSR[VRT+32].word[0]		VSR[VRT+32].word[1]		VSR[VRT+32].word[2]		VSR[VRT+32].word[3]	
	0	16	32	48	64	80	96	112 127

Vector Polynomial Multiply-Sum Word VX-form

vpmsumw VRT,VRA,VRB

4	VRT	VRA	VRB	1160
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
   prod[i].bit[0:62] ← 0
   srcA ← VSR[VRA+32].word[i]
   srcB ← VSR[VRB+32].word[i]
   do j = 0 to 31
      do k = 0 to j
         gbit ← srcA.bit[k] & srcB.bit[j-k]
         prod[i].bit[j] ← prod[i].bit[j] ⊕ gbit
      end
   end
   do j = 32 to 62
      do k = j-31 to 31
         gbit ← (srcA.bit[k] & srcB.bit[j-k])
         prod[i].bit[j] ← prod[i].bit[j] ⊕ gbit
      end
   end
end
VSR[VRT+32].dword[0] ← 0b0 || (prod[0] ⊕ prod[1])
VSR[VRT+32].dword[1] ← 0b0 || (prod[2] ⊕ prod[3])
```

For each integer value `i` from 0 to 3, do the following.

Let `prod[i]` be the 63-bit result of a binary polynomial multiplication of the contents of word element `i` of `VSR[VRA+32]` and the contents of word element `i` of `VSR[VRB+32]`.

For each integer value `i` from 0 to 1, do the following.

The exclusive-OR of `prod[2×i]` and `prod[2×i+1]` is placed in bits 1:63 of doubleword element `i` of `VSR[VRT+32]`. Bit 0 of doubleword element `i` of `VSR[VRT+32]` is set to 0.

Special Registers Altered:

None

Vector Polynomial Multiply-Sum Doubleword VX-form

vpmsumd VRT,VRA,VRB

4	VRT	VRA	VRB	1224
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 1
   prod[i].bit[0:126] ← 0
   srcA ← VSR[VRA+32].doubleword[i]
   srcB ← VSR[VRB+32].doubleword[i]
   do j = 0 to 63
      do k = 0 to j
         gbit ← srcA.bit[k] & srcB.bit[j-k]
         prod[i].bit[j] ← prod[i].bit[j] ⊕ gbit
      end
   end
   do j = 64 to 126
      do k = j-63 to 63
         gbit ← (srcA.bit[k] & srcB.bit[j-k])
         prod[i].bit[j] ← prod[i].bit[j] ⊕ gbit
      end
   end
end
VSR[VRT+32] ← 0b0 || (prod[0] ⊕ prod[1])
```

Let `prod[0]` be the 127-bit result of a binary polynomial multiplication of the contents of doubleword element 0 of `VSR[VRA+32]` and the contents of doubleword element 0 of `VSR[VRB+32]`.

Let `prod[1]` be the 127-bit result of a binary polynomial multiplication of the contents of doubleword element 1 of `VSR[VRA+32]` and the contents of doubleword element 1 of `VSR[VRB+32]`.

The exclusive-OR of `prod[0]` and `prod[1]` is placed in bits 1:127 of `VSR[VRT+32]`. Bit 0 of `VSR[VRT+32]` is set to 0.

Special Registers Altered:

None

Register Data Layout for vpmsumw

src1	VSR[VRA+32].word[0]	VSR[VRA+32].word[1]	VSR[VRA+32].word[2]	VSR[VRA+32].word[3]
src2	VSR[VRB+32].word[0]	VSR[VRB+32].word[1]	VSR[VRB+32].word[2]	VSR[VRB+32].word[3]

result	VSR[VRT+32].dword[0]		VSR[VRT+32].dword[1]	
	0	32	64	96 127

Register Data Layout for vpmsumd

src1	VSR[VRA+32].dword[0]	VSR[VRA+32].dword[1]
src2	VSR[VRB+32].dword[0]	VSR[VRB+32].dword[1]

result	VSR[VRT+32]		
	0	64	127

6.11.4 Vector Permute & Exclusive-OR Instruction

Vector Permute & Exclusive-OR VA-form

vpermxor VRT,VRA,VRB,VRC

4	VRT	VRA	VRB	VRC	45
0	6	11	16	21	26 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 15
   indexA ← VSR[VRC+32].byte[i].bit[0:3]
   indexB ← VSR[VRC+32].byte[i].bit[4:7]
   src1   ← VSR[VRA+32].byte[indexA]
   src2   ← VSR[VRB+32].byte[indexB]
   VSR[VRT+32].byte[i] ← src1 ⊕ src2
end
```

For each integer value `i` from 0 to 15, do the following.

Let `indexA` be the contents of bits 0:3 of byte element `i` of `VSR[VRC+32]`.
Let `indexB` be the contents of bits 4:7 of byte element `i` of `VSR[VRC+32]`.

The exclusive OR of the contents of byte element `indexA` of `VSR[VRA+32]` and the contents of byte element `indexB` of `VSR[VRB+32]` is placed into byte element `i` of `VSR[VRT+32]`.

Special Registers Altered:

None

Register Data Layout for vpermxor

	.byte[0]	.byte[1]	.byte[2]	.byte[3]	.byte[4]	.byte[5]	.byte[6]	.byte[7]	.byte[8]	.byte[9]	.byte[10]	.byte[11]	.byte[12]	.byte[13]	.byte[14]	.byte[15]
src1																
src2																
src3																
result																

0 8 16 24 32 40 48 56 64 72 80 88 96 104 112 120 127

6.12 Vector Bit Manipulation Instructions

6.12.1 Vector Gather Bits Instructions

Vector Gather Bits by Bytes by Doubleword VX-form

vgbbd VRT,VRB

4	VRT	///	VRB	1292
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 1
  do j = 0 to 7
    do k = 0 to 7
      b ← VSR[VRB+32].dword[i].byte[k].bit[j]
      VSR[VRT+32].dword[i].byte[j].bit[k] ← b
    end
  end
end
```

Let src be the contents of VSR[VRB+32], composed of two doubleword elements numbered 0 and 1.

Let each doubleword element be composed of eight bytes numbered 0 through 7.

An 8-bit × 8-bit bit-matrix transpose is performed on the contents of each doubleword element of VSR[VRB+32] (see Figure 6.8).

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 VSR[VRB+32] are concatenated and placed into byte 0 of doubleword element i of VSR[VRT+32].

The contents of bit 1 of each byte of doubleword element i of VSR[VRB+32] are concatenated and placed into byte 1 of doubleword element i of VSR[VRT+32].

The contents of bit 2 of each byte of doubleword element i of VSR[VRB+32] are concatenated and placed into byte 2 of doubleword element i of VSR[VRT+32].

The contents of bit 3 of each byte of doubleword element i of VSR[VRB+32] are concatenated and placed into byte 3 of doubleword element i of VSR[VRT+32].

The contents of bit 4 of each byte of doubleword element i of VSR[VRB+32] are concatenated and placed into byte 4 of doubleword element i of VSR[VRT+32].

The contents of bit 5 of each byte of doubleword element i of VSR[VRB+32] are concatenated and placed into byte 5 of doubleword element i of VSR[VRT+32].

The contents of bit 6 of each byte of doubleword element i of VSR[VRB+32] are concatenated and placed into byte 6 of doubleword element i of VSR[VRT+32].

The contents of bit 7 of each byte of doubleword element i of VSR[VRB+32] are concatenated and placed into byte 7 of doubleword element i of VSR[VRT+32].

Special Registers Altered:

None

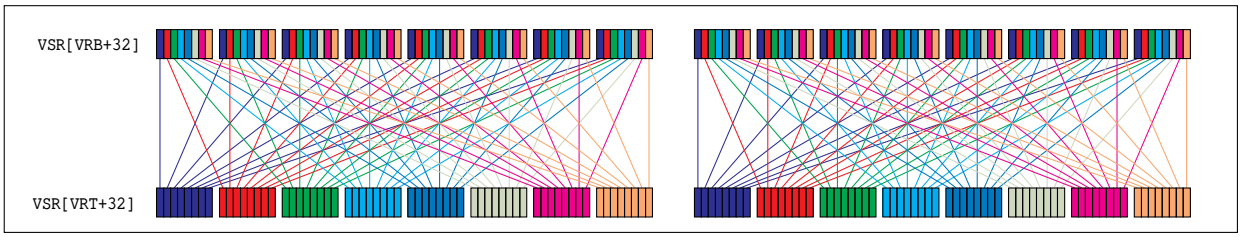
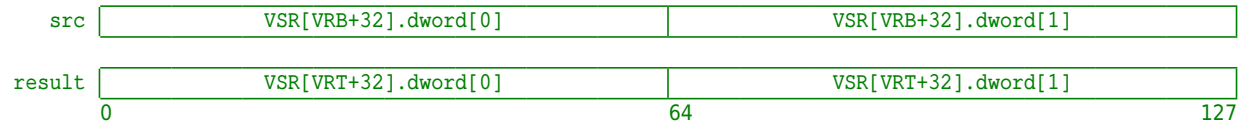


Figure 6.8: Vector Gather Bits by Bytes by Doubleword

Register Data Layout for vgbbd



Vector Gather every Nth Bit VX-form

vgnb RT,VRB,N

4	RT	//	N	VRB	1228
0	6	11	13	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

if N<2 | N>7 then
   result ← UNDEFINED
else do
   j ← 0
   result ← 0x0000_0000_0000_0000
   do i = 0 to 127 by N
      result.bit[j] ← VSR[VRB+32].bit[i]
      j ← j + 1
   end
end
GPR[RT] ← result
```

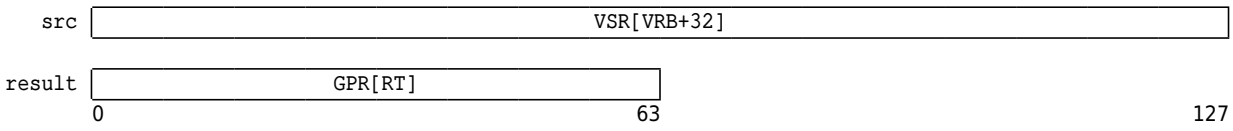
Starting with bit 0, the contents of every N^{th} bit of VSR[VRB+32] are concatenated and placed into GPR[RT].

N can be any value between 2 and 7, inclusive.

Special Registers Altered:

None

Register Data Layout for vgnb



6.12.2 Vector Count Leading Zeros Instructions

Vector Count Leading Zeros Byte VX-form

vclzb VRT,VRB

4	VRT	///	VRB	1794
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 15
   n ← 0
   do while n < 8
      if VSR[VRB+32].byte[i].bit[n] = 0b1 then
         leave
      n ← n + 1
   end
   VSR[VRT+32].byte[i] ← n
end
```

For each integer value `i` from 0 to 15, do the following.

A count of the number of consecutive zero bits starting at bit 0 of byte element `i` of `VSR[VRB+32]` is placed into byte element `i` of `VSR[VRT+32]`. This number ranges from 0 to 8, inclusive.

Special Registers Altered:

None

Vector Count Leading Zeros Halfword VX-form

vclzh VRT,VRB

4	VRT	///	VRB	1858
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 7
   n  ← 0
   do while n < 16
      if VSR[VRB+32].hword[i].bit[n] = 0b1 then
         leave
      n  ← n + 1
   end
   VSR[VRT+32].hword[i] ← n
end
```

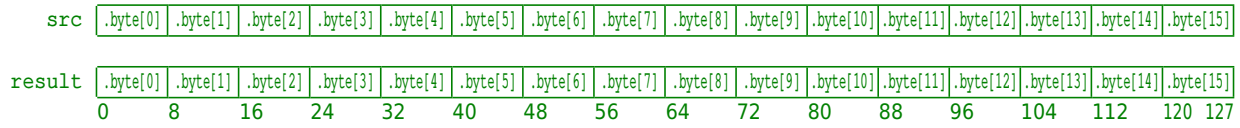
For each integer value `i` from 0 to 7, do the following.

A count of the number of consecutive zero bits starting at bit 0 of halfword element `i` of `VSR[VRB+32]` is placed into halfword element `i` of `VSR[VRT+32]`. This number ranges from 0 to 16, inclusive.

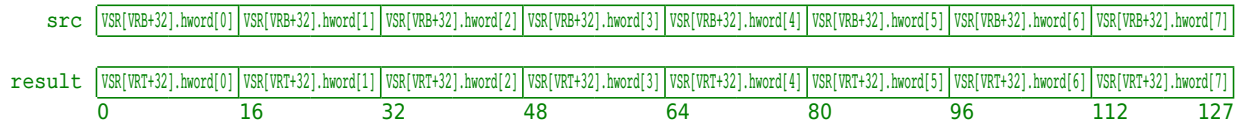
Special Registers Altered:

None

Register Data Layout for vclzb



Register Data Layout for vclzh



Vector Count Leading Zeros Word VX-form

vclzw VRT,VRB

4	VRT	///	VRB	1922
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
   n  ← 0
   do while n < 32
      if VSR[VRB+32].word[i].bit[n] = 0b1 then
         leave
      n  ← n + 1
   end
   VSR[VRT+32].word[i] ← n
end
```

For each integer value `i` from 0 to 3, do the following.

A count of the number of consecutive zero bits starting at bit 0 of word element `i` of `VSR[VRB+32]` is placed into word element `i` of `VSR[VRT+32]`. This number ranges from 0 to 32, inclusive.

Special Registers Altered:

None

Register Data Layout for vclzw

src	VSR[VRB+32].word[0]	VSR[VRB+32].word[1]	VSR[VRB+32].word[2]	VSR[VRB+32].word[3]
result	VSR[VRT+32].word[0]	VSR[VRT+32].word[1]	VSR[VRT+32].word[2]	VSR[VRT+32].word[3]
	0	32	64	96 127

Vector Count Leading Zeros Doubleword VX-form

vclzd VRT,VRB

4	VRT	///	VRB	1986
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 1
   n ← 0
   do while (n<64) & (VSR[VRB+32].dword[i].bit[n]=0b0)
      n ← n + 1
   end
   VSR[VRT+32].dword[i] ← n
end
```

For each integer value `i` from 0 to 1, do the following.

A count of the number of consecutive zero bits starting at bit 0 of doubleword element `i` of `VSR[VRB+32]` is placed into doubleword element `i` of `VSR[VRT+32]`. This number ranges from 0 to 64, inclusive.

Special Registers Altered:

None

Vector Count Leading Zeros Doubleword under bit Mask VX-form

vclzdm VRT,VRA,VRB

4	VRT	VRA	VRB	1924
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 1
   count ← 0
   do j = 0 to 63
      if VSR[VRB+32].dword[i].bit[j]=1 then do
         if VSR[VRA+32].dword[i].bit[i]=1 then
            break
         count ← count + 1
      end
   end
   VSR[VRT+32].dword[i] ← EXTZ64(count)
end
```

For each integer value `i` from 0 to 1, starting on the left, count the number of consecutive 0 bits in doubleword element `i` of `VSR[VRA+32]` corresponding to 1 bits in doubleword element `i` of `VSR[VRB+32]`. Place count in doubleword element `i` of `VSR[VRT+32]`.

Special Registers Altered:

None

Register Data Layout for vclzd

src	VSR[VRB+32].dword[0]	VSR[VRB+32].dword[1]
result	VSR[VRT+32].dword[0]	VSR[VRT+32].dword[1]
	0 64	127

Register Data Layout for vclzdm

src1	VSR[VRA+32].dword[0]	VSR[VRA+32].dword[1]
src2	VSR[VRB+32].dword[0]	VSR[VRB+32].dword[1]
result	VSR[VRT+32].dword[0]	VSR[VRT+32].dword[1]
	0 64	127

6.12.3 Vector Count Trailing Zeros Instructions

Vector Count Trailing Zeros Byte VX-form

vctzb VRT,VRB

4	VRT	28	VRB	1538
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 15
   n ← 0
   do while n < 8
      if VSR[VRB+32].byte[i].bit[7-n] = 0b1 then
         leave
      n ← n + 1
   end
   VSR[VRT+32].byte[i] ← CHOP8(EXTZ(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 7 of byte element `i` of `VSR[VRB+32]` is placed into byte element `i` of `VSR[VRT+32]`. This number ranges from 0 to 8, inclusive.

Special Registers Altered:

None

Vector Count Trailing Zeros Halfword VX-form

vctzh VRT,VRB

4	VRT	29	VRB	1538
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 7
   n  ← 0
   do while n < 16
      if VSR[VRB+32].hword[i].bit[15-n] = 0b1 then
         leave
      n  ← n + 1
   end
   VSR[VRT+32].hword[i] ← CHOP16(EXTZ(n))
end
```

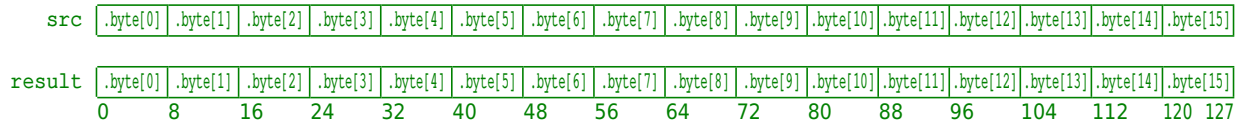
For each integer value `i` from 0 to 7, do the following.

A count of the number of consecutive zero bits starting at bit 15 of halfword element `i` of `VSR[VRB+32]` is placed into halfword element `i` of `VSR[VRT+32]`. This number ranges from 0 to 16, inclusive.

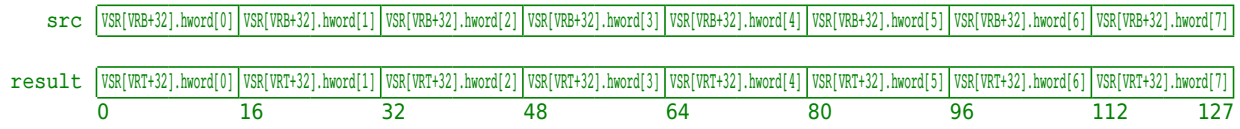
Special Registers Altered:

None

Register Data Layout for vctzb



Register Data Layout for vctzh



Vector Count Trailing Zeros Word VX-form

vctzw VRT,VRB

4	VRT	30	VRB	1538
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
   n  ← 0
   do while n < 32
      if VSR[VRB+32].word[i].bit[31-n] = 0b1 then
         leave
      n  ← n + 1
   end
   VSR[VRT+32].word[i] ← CHOP32(EXTZ(n))
end
```

For each integer value `i` from 0 to 3, do the following.

A count of the number of consecutive zero bits starting at bit 31 of word element `i` of `VSR[VRB+32]` is placed into word element `i` of `VSR[VRT+32]`. This number ranges from 0 to 32, inclusive.

Special Registers Altered:

None

Register Data Layout for vctzw

src	VSR[VRB+32].word[0]	VSR[VRB+32].word[1]	VSR[VRB+32].word[2]	VSR[VRB+32].word[3]
result	VSR[VRT+32].word[0]	VSR[VRT+32].word[1]	VSR[VRT+32].word[2]	VSR[VRT+32].word[3]
	0	32	64	96 127

Vector Count Trailing Zeros Doubleword VX-form

vctzd VRT,VRB

4	VRT	31	VRB	1538
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 1
   n  ← 0
   do while n < 64
      if VSR[VRB+32].dword[i].bit[63-n] = 0b1 then
         leave
      n  ← n + 1
   end
   VSR[VRT+32].dword[i] ← CHOP64(EXTZ(n))
end
```

For each integer value `i` from 0 to 1, do the following.

A count of the number of consecutive zero bits starting at bit 63 of doubleword element `i` of `VSR[VRB+32]` is placed into doubleword element `i` of `VSR[VRT+32]`. This number ranges from 0 to 64, inclusive.

Special Registers Altered:

None

Vector Count Trailing Zeros Doubleword under bit Mask VX-form

vctzdm VRT,VRA,VRB

4	VRT	VRA	VRB	1988
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 1
   count ← 0
   do j = 0 to 63
      if VSR[VRB+32].dword[i].bit[63-j]=1 then do
         if VSR[VRA+32].dword[i].bit[63-i]=1 then
            break
         count ← count + 1
      end
   end
   VSR[VRT+32].dword[i] ← EXTZ64(count)
end
```

For each integer value `i` from 0 to 1, starting on the right, count the number of consecutive 0 bits in doubleword element `i` of `VSR[VRA+32]` corresponding to 1 bits in doubleword element `i` of `VSR[VRB+32]`. Place count in doubleword element `i` of `VSR[VRT+32]`.

Special Registers Altered:

None

Register Data Layout for vctzd

src	VSR[VRB+32].dword[0]	VSR[VRB+32].dword[1]
result	VSR[VRT+32].dword[0]	VSR[VRT+32].dword[1]
	0 64	127

Register Data Layout for vctzdm

src1	VSR[VRA+32].dword[0]	VSR[VRA+32].dword[1]
src2	VSR[VRB+32].dword[0]	VSR[VRB+32].dword[1]
result	VSR[VRT+32].dword[0]	VSR[VRT+32].dword[1]
	0 64	127

6.12.4 Vector Count Leading/Trailing Zero LSB Instructions

Vector Count Leading Zero Least-Significant Bits Byte VX-form

vclzlsbb RT,VRB

4	RT	0	VRB	1538
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

count ← 0
do while count < 16
   if VSR[VRB+32].byte[count].bit[7]=1 then
      break
   count ← count + 1
end
GPR[RT] ← EXTZ64(count)
```

Let count be the number of contiguous leading byte elements in VSR[VRB+32] having a zero least-significant bit.

count is placed into GPR[RT].

Special Registers Altered:

None

Vector Count Trailing Zero Least-Significant Bits Byte VX-form

vctzlsbb RT,VRB

4	RT	1	VRB	1538
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

count ← 0
do while count < 16
   if VSR[VRB+32].byte[15-count].bit[7]=1 then
      break
   count ← count + 1
end
GPR[RT] ← EXTZ64(count)
```

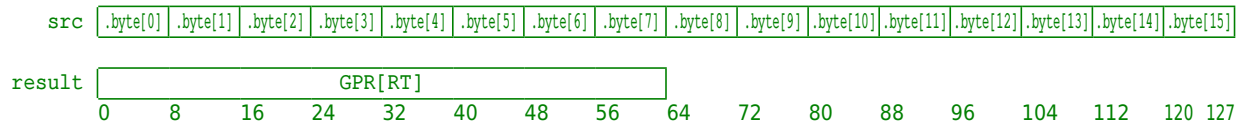
Let count be the number of contiguous trailing byte elements of VSR[VRB+32] having a zero least-significant bit.

count is placed into GPR[RT].

Special Registers Altered:

None

Register Data Layout for vclzlsbb & vctzlsbb



6.12.5 Vector Bit Insert/Extract Instructions

Vector Parallel Bits Deposit Doubleword VX-form

vpdepd VRT,VRA,VRB

4	VRT	VRA	VRB	1485
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 1
  VSR[VRT+32].dword[i] ← 0
  m ← 0
  k ← 0
  do while(m < 64)
    if VSR[VRB+32].dword[i].bit[63-m]=1 then do
      result ← VSR[VRA+32].dword[i].bit[63-k]
      VSR[VRT+32].dword[i].bit[63-m] ← result
      k ← k + 1
    end
    m ← m + 1
  end
end
```

For each integer value i from 0 to 1, do the following.

Let n be the number of bits in doubleword element i of VSR[VRB+32] that contain a 1.

The contents of the rightmost n bits of doubleword element i of VSR[VRA+32] are placed into doubleword element i of VSR[VRT+32] under control of the mask in doubleword element i of VSR[VRB+32] as follows.

- The contents of bit 63 of doubleword element i of VSR[VRA+32] are placed into the bit in doubleword element i of VSR[VRT+32] corresponding to the rightmost bit in doubleword element i of VSR[VRB+32] that contains a 1 (if any),
- the contents of bit 62 of doubleword element i of VSR[VRA+32] are placed into the bit in doubleword element i of VSR[VRT+32] corresponding to the second rightmost bit in doubleword element i of VSR[VRB+32] that contains a 1 (if any), and so forth until
- the contents of bit 64−n of doubleword element i of VSR[VRA+32] are placed into the bit in doubleword element i of VSR[VRT+32] corresponding to the leftmost bit in doubleword element i of VSR[VRB+32] that contains a 1 (if any).

The contents of bits in doubleword element i of VSR[VRT+32] corresponding to bits in doubleword element i of VSR[VRB+32] that contain a 0 are set to 0.

Special Registers Altered:

None

Register Data Layout for vpdepd

src1	VSR[VRA+32].dword[0]	VSR[VRA+32].dword[1]
src2	VSR[VRB+32].dword[0]	VSR[VRB+32].dword[1]

result	VSR[VRT+32].dword[0]	VSR[VRT+32].dword[1]
	0 64	127

Vector Parallel Bits Extract Doubleword VX-form

vpextd VRT,VRA,VRB

4	VRT	VRA	VRB	1421
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 1
  result ← 0
  m ← 0
  k ← 0
  do while(m < 64)
    if VSR[VRB+32].dword[i].bit[63-m]=1 then do
      result.bit[63-k] ←
         VSR[VRA+32].dword[i].bit[63-m]
      k ← k + 1
    end
    m ← m + 1
  end
  VSR[VRT+32].dword[i] ← result
end
```

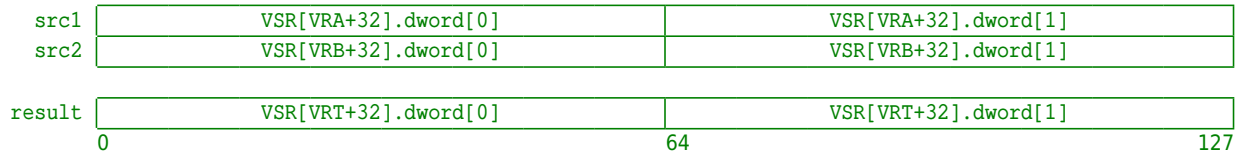
For each integer value `i` from 0 to 1, do the following.

Starting from the right, for each bit in doubleword element `i` of `VSR[VRB+32]` that is equal to 1, place the contents of the corresponding bit in doubleword element `i` of `VSR[VRA+32]` into the rightmost unoccupied bit of doubleword element `i` of `VSR[VRT+32]`. Any bits in doubleword element `i` of `VSR[VRT+32]`to the left of the most-significant bit copied are set to 0.

Special Registers Altered:

None

Register Data Layout for vpextd



6.12.6 Vector Centrifuge Instruction

Vector Centrifuge Doubleword VX-form

vcfuged VRT,VRA,VRB

4	VRT	VRA	VRB	1357
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 1
  ptr0 ← 0
  ptr1 ← 0
  do j = 0 to 63
    if VSR[VRB+32].dword[i].bit[j]=0b0 then do
      result.bit[ptr0] ←
         VSR[VRA+32].dword[i].bit[j]
      ptr0 ← ptr0 + 1
    end
    if VSR[VRB+32].dword[i].bit[63-j]=1 then do
      result.bit[63-ptr1] ←
         VSR[VRA+32].dword[i].bit[63-j]
      ptr1 ← ptr1 + 1
    end
  end
  VSR[VRT+32].dword[i] ← result
end
```

For each doubleword element `i` of `VSR[VRA+32]`, the bits whose corresponding bits in the mask in doubleword element `i` of `VSR[VRB+32]` equal 1 are placed in the rightmost bits in doubleword element `i` of `VSR[VRT+32]`, maintaining their original relative order. The other bits in doubleword element `i` of `VSR[VRA+32]` are placed in the leftmost bits in doubleword element `i` of `VSR[VRT+32]`, maintaining their original relative order.

Special Registers Altered:

None

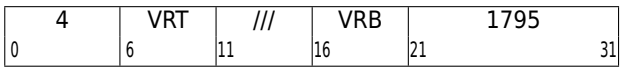
Register Data Layout for vcfuged

src1	VSR[VRA+32].dword[0]	VSR[VRA+32].dword[1]
src2	VSR[VRB+32].dword[0]	VSR[VRB+32].dword[1]
result	VSR[VRT+32].dword[0]	VSR[VRT+32].dword[1]
	0 64	127

6.12.7 Vector Population Count Instructions

Vector Population Count Byte VX-form

vpopcntb VRT,VRB




```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 15
   n ← 0
   do j = 0 to 7
      n ← n + VSR[VRB+32].byte[i].bit[j]
   end
   VSR[VRT+32].byte[i] ← n
end
```

For each integer value `i` from 0 to 15, do the following.

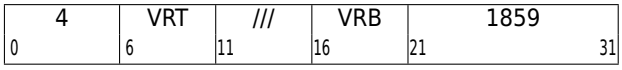
A count of the number of bits set to 1 in byte element `i` of `VSR[VRB+32]` is placed into byte element `i` of `VSR[VRT+32]`. This number ranges from 0 to 8, inclusive.

Special Registers Altered:

None

Vector Population Count Halfword VX-form

vpopcnth VRT,VRB




```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 7
   n ← 0
   do j = 0 to 15
      n ← n + VSR[VRB+32].hword[i].bit[j]
   end
   VSR[VRT+32].hword[i] ← n
end
```

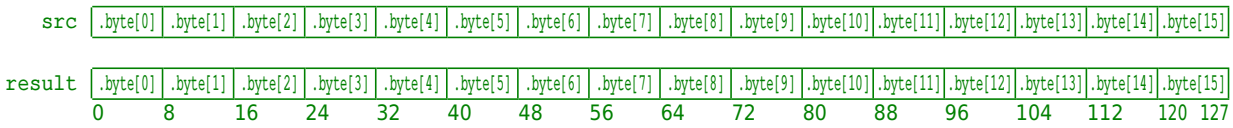
For each integer value `i` from 0 to 7, do the following.

A count of the number of bits set to 1 in halfword element `i` of `VSR[VRB+32]` is placed into halfword element `i` of `VSR[VRT+32]`. This number ranges from 0 to 16, inclusive.

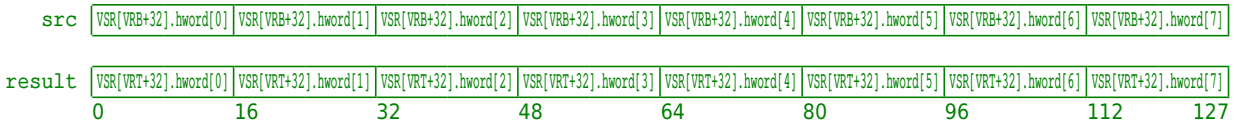
Special Registers Altered:

None

Register Data Layout for vpopcntb



Register Data Layout for vpopcnth



Vector Population Count Word VX-form

vpopcntw VRT,VRB

4	VRT	///	VRB	1923
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
   n ← 0
   do j = 0 to 31
      n ← n + VSR[VRB+32].word[i].bit[j]
   end
   VSR[VRT+32].word[i] ← n
end
```

For each integer value `i` from `0` to `3`, do the following.

A count of the number of bits set to `1` in word element `i` of `VSR[VRB+32]` is placed into word element `i` of `VSR[VRT+32]`. This number ranges from `0` to `32`, inclusive.

Special Registers Altered:

None

Vector Population Count Doubleword VX-form

vpopcntd VRT,VRB

4	VRT	///	VRB	1987
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 1
   n ← 0
   do j = 0 to 63
      n ← n + VSR[VRB+32].dword[i].bit[j]
   end
   VSR[VRT+32].dword[i] ← n
end
```

For each integer value `i` from `0` to `1`, do the following.

A count of the number of bits set to 1 in doubleword element `i` of `VSR[VRB+32]` is placed into doubleword element `i` of `VSR[VRT+32]`. This number ranges from `0` to `64`, inclusive.

Special Registers Altered:

None

Register Data Layout for vpopcntw

src	VSR[VRB+32].word[0]	VSR[VRB+32].word[1]	VSR[VRB+32].word[2]	VSR[VRB+32].word[3]
result	VSR[VRT+32].word[0]	VSR[VRT+32].word[1]	VSR[VRT+32].word[2]	VSR[VRT+32].word[3]
	0	32	64	96 127

Register Data Layout for vpopcntd

src	VSR[VRB+32].dword[0]	VSR[VRB+32].dword[1]
result	VSR[VRT+32].dword[0]	VSR[VRT+32].dword[1]
	0	64 127

6.12.8 Vector Parity Byte Instructions

Vector Parity Byte Word VX-form

vprtybw VRT,VRB

4	VRT	8	VRB	1538
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
  s ← 0
  do j = 0 to 3
    s ← s ⊕ VSR[VRB+32].word[i].byte[j].bit[7]
  end
  VSR[VRT+32].word[i] ← CHOP32(EXTZ(s))
end
```

For each integer value `i` from 0 to 3, do the following

If the sum of the least significant bit in each byte sub-element of word element `i` of `VSR[VRB+32]` is odd, the value 1 is placed into word element `i` of `VSR[VRT+32]`; otherwise the value 0 is placed into word element `i` of `VSR[VRT+32]`.

Special Registers Altered:

None

Vector Parity Byte Doubleword VX-form

vprtybd VRT,VRB

4	VRT	9	VRB	1538
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 1
  s ← 0
  do j = 0 to 7
    s ← s ⊕ VSR[VRB+32].dword[i].byte[j].bit[7]
  end
  VSR[VRT+32].dword[i] ← CHOP64(EXTZ(s))
end
```

For each integer value `i` from 0 to 1, do the following

If the sum of the least significant bit in each byte sub-element of doubleword element `i` of `VSR[VRB+32]` is odd, the value 1 is placed into doubleword element `i` of `VSR[VRT+32]`; otherwise the value 0 is placed into doubleword element `i` of `VSR[VRT+32]`.

Special Registers Altered:

None

Register Data Layout for vprtybw

src1	VSR[VRA+32].word[0]	VSR[VRA+32].word[1]	VSR[VRA+32].word[2]	VSR[VRA+32].word[3]
src2	VSR[VRB+32].word[0]	VSR[VRB+32].word[1]	VSR[VRB+32].word[2]	VSR[VRB+32].word[3]
result	VSR[VRT+32].word[0]	VSR[VRT+32].word[1]	VSR[VRT+32].word[2]	VSR[VRT+32].word[3]
	0	32	64	96 127

Register Data Layout for vprtybd

src1	VSR[VRA+32].dword[0]	VSR[VRA+32].dword[1]
src2	VSR[VRB+32].dword[0]	VSR[VRB+32].dword[1]
result	VSR[VRT+32].dword[0]	VSR[VRT+32].dword[1]
	0	64 127

Vector Parity Byte Quadword VX-form

vprtybq VRT,VRB

4	VRT	10	VRB	1538
0	6	11	16	21 31

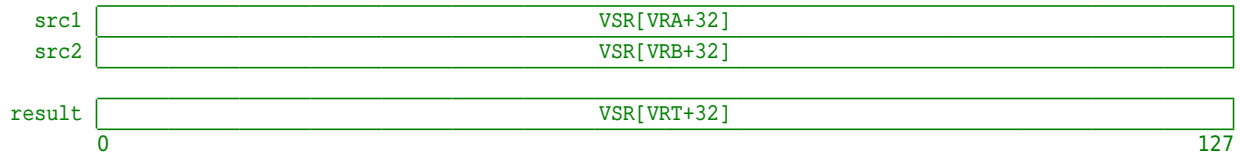
```
if MSR.VEC=0 then Vector_Unavailable()

s ← 0
do j = 0 to 15
   s ← s ⊕ VSR[VRB+32].byte[j].bit[7]
end
VSR[VRT+32] ← CHOP128(EXTZ(s))
```

If the sum of the least significant bit in each byte element of VSR[VRB+32] is odd, the value 1 is placed into VSR[VRT+32]; otherwise the value 0 is placed into VSR[VRT+32].

Special Registers Altered:

None

Register Data Layout for vprtybq



6.12.9 Vector Bit Permute Instructions

Vector Bit Permute Doubleword VX-form

vbpermd VRT,VRA,VRB

4	VRT	VRA	VRB	1484
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 1
   do j = 0 to 7
      index ← VSR[VRB+32].dword[i].byte[j]
      if index < 64 then
         perm.bit[j] ← VSR[VRA+32].dword[i].bit[index]
      else
         perm.bit[j] ← 0
   end
   VSR[VRT+32].dword[i] ← EXTZ64(perm)
end
```

For each integer value `i` from 0 to 1, and for each integer value `j` from 0 to 7, do the following.

Let `index` be the contents of byte sub-element `j` of doubleword element `i` of `VSR[VRB+32]`.

If `index` is less than 64, then the contents of bit `index` of doubleword `i` of `VSR[VRA+32]` are placed into bit `56+j` of doubleword element `i` of `VSR[VRT+32]`. Otherwise, bit `56+j` of doubleword element `i` of `VSR[VRT+32]` is set to 0.

The contents of bits 0:55 of doubleword element `i` of `VSR[VRT+32]` are set to 0.

Special Registers Altered:

None

Register Data Layout for vbpermd

src1	VSR[VRA+32].dword[0]	VSR[VRA+32].dword[1]
src2	VSR[VRB+32].dword[0]	VSR[VRB+32].dword[1]

result	VSR[VRT+32].dword[0]	VSR[VRT+32].dword[1]
	0 64	127

Vector Bit Permute Quadword VX-form

vbpermq VRT,VRA,VRB

4	VRT	VRA	VRB	1356
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 15
   index ← VSR[VRB+32].byte[i]
   if index < 128 then
      perm.bit[i] ← VSR[VRA+32].bit[index]
   else
      perm.bit[i] ← 0
end
VSR[VRT+32].dword[0] ← CHOP64(EXTZ(perm))
VSR[VRT+32].dword[1] ← 0x0000_0000_0000_0000
```

For each integer value `i` from `0` to `15`, do the following.

Let `index` be the contents of byte element `i` of `VSR[VRB+32]`.

If `index` is less than `128`, then the contents of bit `index` of `VSR[VRA+32]` are placed into bit `48+i` of doubleword element `i` of `VSR[VRT+32]`. Otherwise, bit `48+i` of doubleword element `i` of `VSR[VRT+32]` is set to `0`.

The contents of bits `0:47` of `VSR[VRT+32]` are set to `0`.
The contents of bits `64:127` of `VSR[VRT+32]` are set to `0`.

Special Registers Altered:

None

Register Data Layout for vbpermq

Programming Note

The fact that the permuted bit is 0 if the corresponding index value exceeds 127 permits the permuted bits to be selected from a 256-bit quantity, using a single index register. For example, assume that the 256-bit quantity `Q`, from which the permuted bits are to be selected, is in registers `v2` (high-order 128 bits of `Q`) and `v3` (low-order 128 bits of `Q`), that the index values are in register `v1`, with each byte of `v1` containing a value in the range 0:255, and that each byte of register `v4` contains the value 128. The following code sequence selects eight permuted bits from `Q` and places them into the low-order byte of `v6`.

```
vbpermq v6,v1,v2        # select from high-order half of Q
vxor    v0,v1,v4        # adjust index values
vbpermq v5,v0,v3        # select from low-order half of Q
vor     v6,v6,v5        # merge the two selections
```

6.13 Vector Mask Manipulation Instructions

6.13.1 Vector Mask Move Instructions

The *Vector Mask Move* instructions support creating a field mask in a VSR from a bit mask specified either in a GPR or as an immediate operand.

Move to VSR Byte Mask VX-form

mtvsrbm VRT,RB

4	VRT	16	RB	1602
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 15
   if GPR[RB].bit[48+i]=0 then
      VSR[VRT+32].byte[i] ← 0x00
   else
      VSR[VRT+32].byte[i] ← 0xFF
end
```

Let bm be the contents of bits 48:63 of GPR[RB].

For each integer value i from 0 to 15, do the following.

The contents of byte element i of VSR[VRT+32] is set to all 0s if bit i of bm is equal to 0.

The contents of byte element i of VSR[VRT+32] is set to all 1s if bit i of bm is equal to 1.

Special Registers Altered:

None

Move to VSR Halfword Mask VX-form

mtvsrhm VRT,RB

4	VRT	17	RB	1602
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 7
   if GPR[RB].bit[56+i]=0 then
      VSR[VRT+32].hword[i] ← 0x0000
   else
      VSR[VRT+32].hword[i] ← 0xFFFF
end
```

Let bm be the contents of bits 56:63 of GPR[RB].

For each integer value i from 0 to 7, do the following.

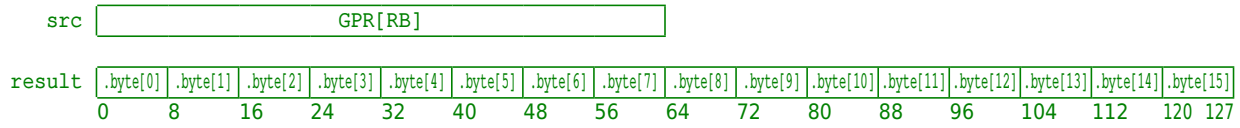
The contents of halfword element i of VSR[VRT+32] is set to all 0s if bit i of bm is equal to 0.

The contents of halfword element i of VSR[VRT+32] is set to all 1s if bit i of bm is equal to 1.

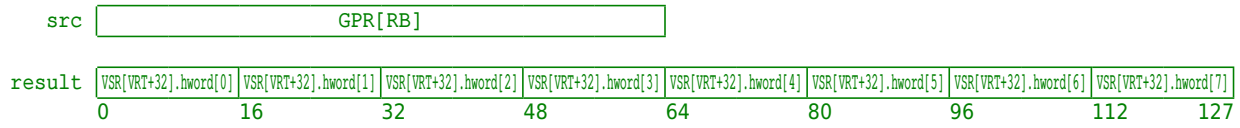
Special Registers Altered:

None

Register Data Layout for mtvsrbm



Register Data Layout for mtvsrhm



Move to VSR Word Mask VX-form

mtvsrwm VRT,RB

4	VRT	18	RB	1602
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
   if GPR[RB].bit[60+i]=0 then
      VSR[VRT+32].word[i] ← 0x0000_0000
   else
      VSR[VRT+32].word[i] ← 0xFFFF_FFFF
end
```

Let bm be the contents of bits 60:63 of GPR[RB].

For each integer value i from 0 to 3, do the following.

The contents of word element i of VSR[VRT+32] is set to all 0s if bit i of bm is equal to 0.

The contents of word element i of VSR[VRT+32] is set to all 1s if bit i of bm is equal to 1.

Special Registers Altered:

None

Move to VSR Doubleword Mask VX-form

mtvsrdm VRT,RB

4	VRT	19	RB	1602
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 1
   if GPR[RB].bit[62+i]=0 then
      VSR[VRT+32].dword[i] ← 0x0000_0000_0000_0000
   else
      VSR[VRT+32].dword[i] ← 0xFFFF_FFFF_FFFF_FFFF
end
```

Let bm be the contents of bits 62:63 of GPR[RB].

For each integer value i from 0 to 1, do the following.

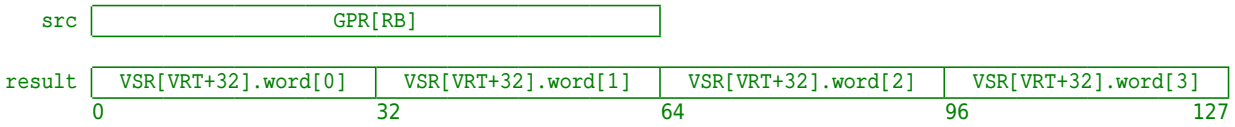
The contents of doubleword element i of VSR[VRT+32] is set to all 0s if bit i of bm is equal to 0.

The contents of doubleword element i of VSR[VRT+32] is set to all 1s if bit i of bm is equal to 1.

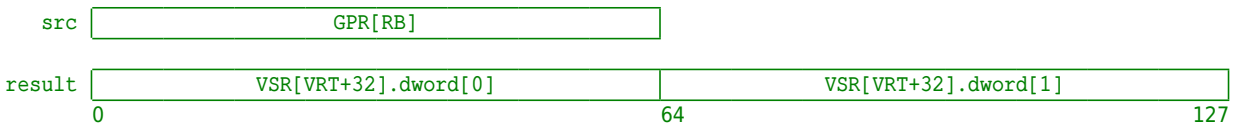
Special Registers Altered:

None

Register Data Layout for mtvsrwm

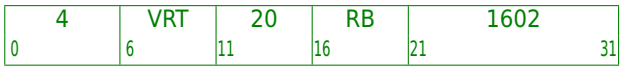


Register Data Layout for mtvsrdm



Move to VSR Quadword Mask VX-form

mtvsrqm VRT,RB




```
if MSR.VEC=0 then Vector_Unavailable()

if GPR[RB].bit[63]=0 then
   VSR[VRT+32] ←
      0x0000_0000_0000_0000_0000_0000_0000_0000
else
   VSR[VRT+32] ←
      0xFFFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF
```

Let bm be the contents of bits 63 of GPR[RB].

The contents of VSR[VRT+32] is set to all 0s if bm is equal to 0.

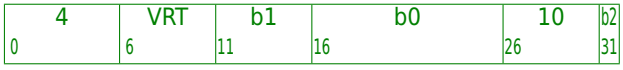
The contents of VSR[VRT+32] is set to all 1s if bm is equal to 1.

Special Registers Altered:

None

Move To VSR Byte Mask Immediate DX-form

mtvsrbmi VRT,bm




```
if MSR.VEC=0 then Vector_Unavailable()

bm.bit[0:9]   ← b0
bm.bit[10:14] ← b1
bm.bit[15]    ← b2
do i = 0 to 15
   if bm.bit[i]=0 then
      VSR[VRT+32].byte[i] ← 0x00
   else
      VSR[VRT+32].byte[i] ← 0xFF
end
```

Let bm be the concatenation of b0, b1 and b2.

For each integer value i from 0 to 15, do the following.

The contents of byte element i of VSR[VRT+32] is set to all 0s if bit i of bm is equal to 0.

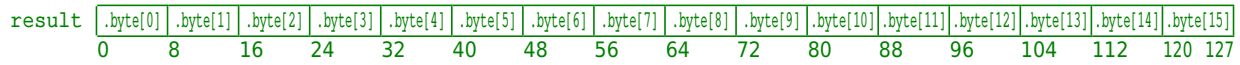
The contents of byte element i of VSR[VRT+32] is set to all 1s if bit i of bm is equal to 1.

Special Registers Altered:

None

Register Data Layout for mtvsrqm

Register Data Layout for mtvsrbmi



6.13.2 Vector Expand Mask Instructions

The *Vector Expand Mask* instructions support creating a field mask by replicating the contents of bit 0 of each element in the source VSR to all bits in the corresponding element in the target VSR.

Vector Expand Byte Mask VX-form

vexpandbm VRT,VRB

4	VRT	0	VRB	1602
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 15
   if VSR[VRB+32].byte[i].bit[0]=1 then
      VSR[VRT+32].byte[i] ← 0xFF
   end
      VSR[VRT+32].byte[i] ← 0x00
end
```

For each integer value `i` from 0 to 15, do the following.

Let `bmi` be the contents of bit 0 of byte element `i` of `VSR[VRB+32]`.

The contents of byte element `i` of `VSR[VRT+32]` are set to all 0s if `bmi` is equal to 0.

The contents of byte element `i` of `VSR[VRT+32]` are set to all 1s if `bmi` is equal to 1.

Special Registers Altered:

None

Vector Expand Halfword Mask VX-form

vexpandhm VRT,VRB

4	VRT	1	VRB	1602
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 7
   if VSR[VRB+32].hword[i].bit[0]=1 then
      VSR[VRT+32].hword[i] ← 0xFFFF
   else
      VSR[VRT+32].hword[i] ← 0x0000
end
```

For each integer value `i` from 0 to 7, do the following.

Let `bmi` be the contents of bit 0 of halfword element `i` of `VSR[VRB+32]`,

The contents of halfword element `i` of `VSR[VRT+32]` are set to all 0s if `bmi` is equal to 0.

The contents of halfword element `i` of `VSR[VRT+32]` are set to all 1s if `bmi` is equal to 1.

Special Registers Altered:

None

Register Data Layout for vexpandbm

	.byte[0]	.byte[1]	.byte[2]	.byte[3]	.byte[4]	.byte[5]	.byte[6]	.byte[7]	.byte[8]	.byte[9]	.byte[10]	.byte[11]	.byte[12]	.byte[13]	.byte[14]	.byte[15]
src																
result																

0 8 16 24 32 40 48 56 64 72 80 88 96 104 112 120 127

Register Data Layout for vexpandhm

	VSR[VRB+32].hword[0]	VSR[VRB+32].hword[1]	VSR[VRB+32].hword[2]	VSR[VRB+32].hword[3]	VSR[VRB+32].hword[4]	VSR[VRB+32].hword[5]	VSR[VRB+32].hword[6]	VSR[VRB+32].hword[7]
src								
result	VSR[VRT+32].hword[0]	VSR[VRT+32].hword[1]	VSR[VRT+32].hword[2]	VSR[VRT+32].hword[3]	VSR[VRT+32].hword[4]	VSR[VRT+32].hword[5]	VSR[VRT+32].hword[6]	VSR[VRT+32].hword[7]

0 16 32 48 64 80 96 112 127

Vector Expand Word Mask VX-form

vexpandwm VRT,VRB

4	VRT	2	VRB	1602
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
   if VSR[VRB+32].word[i].bit[0]=1 then
      VSR[VRT+32].word[i] ← 0xFFFF_FFFF
   else
      VSR[VRT+32].word[i] ← 0x0000_0000
end
```

For each integer value `i` from 0 to 3, do the following.

Let `bmi` be the contents of bit 0 of word element `i` of `VSR[VRB+32]`.

The contents of word element `i` of `VSR[VRT+32]` are set to all 0s if `bmi` is equal to 0.

The contents of word element `i` of `VSR[VRT+32]` are set to all 1s if `bmi` is equal to 1.

Special Registers Altered:

None

Vector Expand Doubleword Mask VX-form

vexpanddm VRT,VRB

4	VRT	3	VRB	1602
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 1
   if VSR[VRB+32].dword[i].bit[0]=1 then
      VSR[VRT+32].dword[i] ← 0xFFFF_FFFF_FFFF_FFFF
   else
      VSR[VRT+32].dword[i] ← 0x0000_0000_0000_0000
end
```

For each integer value `i` from 0 to 1, do the following.

Let `bmi` be the contents of bit 0 of doubleword element `i` of `VSR[VRB+32]`,

The contents of doubleword element `i` of `VSR[VRT+32]` are set to all 0s if `bmi` is equal to 0.

The contents of doubleword element `i` of `VSR[VRT+32]` are set to all 1s if `bmi` is equal to 1.

Special Registers Altered:

None

Register Data Layout for vexpandwm

src	VSR[VRB+32].word[0]	VSR[VRB+32].word[1]	VSR[VRB+32].word[2]	VSR[VRB+32].word[3]
result	VSR[VRT+32].word[0]	VSR[VRT+32].word[1]	VSR[VRT+32].word[2]	VSR[VRT+32].word[3]
	0	32	64	96 127

Register Data Layout for vexpanddm

src	VSR[VRB+32].dword[0]	VSR[VRB+32].dword[1]
result	VSR[VRT+32].dword[0]	VSR[VRT+32].dword[1]
	0	64 127

Vector Expand Quadword Mask VX-form

vexpandqm VRT,VRB

4	VRT	4	VRB	1602
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

if VSR[VRB+32].bit[0]=1 then
   VSR[VRT+32] ←
      0xFFFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF
else
   VSR[VRT+32] ←
      0x0000_0000_0000_0000_0000_0000_0000_0000
```

Let `bmi` be the contents of bit 0 of `VSR[VRB+32]`.

The contents of `VSR[VRT+32]` are set to all 0s if `bmi` is equal to 0.

The contents of `VSR[VRT+32]` are set to all 1s if `bmi` is equal to 1.

Special Registers Altered:

None

Register Data Layout for vexpandqm

6.13.3 Vector Count Mask Bits Instructions

The *Vector Count Mask Bits* instructions count the number of true (or false) mask bits (bit 0 of each element) in a VSR and place the count in the leftmost byte of a GPR (i.e., can be used by *Load VSX Vector with Length* and *Store Vector with Length*).

Vector Count Mask Bits Byte VX-form

vcntmbb RT,VRB,MP


```
if MSR.VEC=0 then Vector_Unavailable()

count = 0
do i = 0 to 15
   count ← count +
      EXTZ8(VSR[VRB+32].byte[i].bit[0]=MP)
end

GPR[RT] ← count << 56
```

The number of byte elements having bit 0 set to the value MP in VSR[VRB+32] is placed into bits 0:7 of GPR[RT]. Bits 8:63 of GPR[RT] are set to 0.

Special Registers Altered:

None

Vector Count Mask Bits Halfword VX-form

vcntmbh RT,VRB,MP

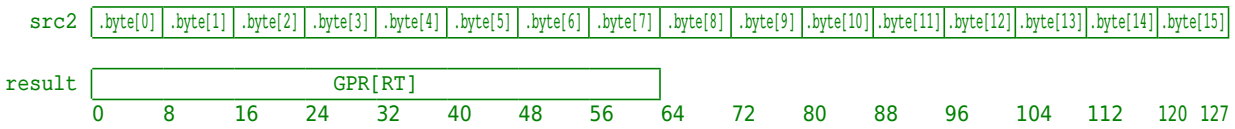

```
if MSR.VEC=0 then Vector_Unavailable()

count = 0
do i = 0 to 7
   count ← count +
      EXTZ64(VSR[VRB+32].hword[i].bit[0]=MP)

GPR[RT] ← count << 57
```

The number of halfword elements having bit 0 set to the value MP in VSR[VRB+32] is placed into bits 0:6 of GPR[RT]. Bits 7:63 of GPR[RT] are set to 0.

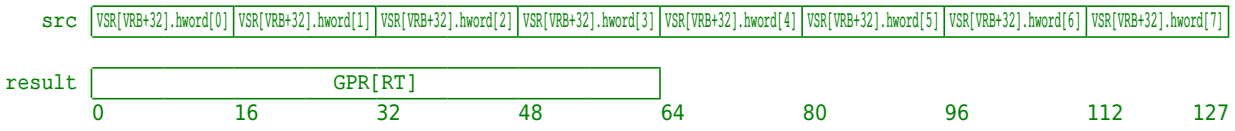
Special Registers Altered:

None

Register Data Layout for vcntmbb



Register Data Layout for vcntmbh



Vector Count Mask Bits Word VX-form

vcntmbw RT,VRB,MP


```
if MSR.VEC=0 then Vector_Unavailable()

count = 0
do i = 0 to 3
   count ← count +
      EXTZ64(VSR[VRB+32].word[i].bit[0]=MP)
end

GPR[RT] ← count << 58
```

The number of word elements having bit 0 set to the value MP in VSR[VRB+32] is placed into bits 0:5 of GPR[RT]. Bits 6:63 of GPR[RT] are set to 0.

Special Registers Altered:

None

Vector Count Mask Bits Doubleword VX-form

vcntmbd RT,VRB,MP

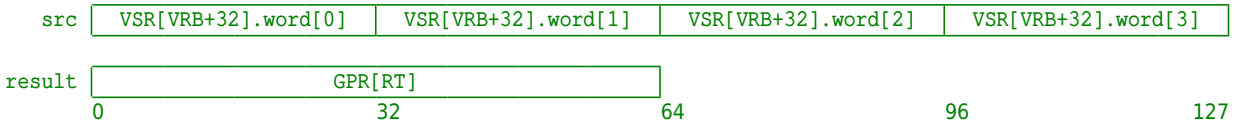

```
if MSR.VEC=0 then Vector_Unavailable()

count = 0
do i = 0 to 1
   count ← count +
      EXTZ64(VSR[VRB+32].dword[i].bit[0]=MP)
end

GPR[RT] ← count << 59
```

The number of doubleword elements having bit 0 set to the value MP in VSR[VRB+32] is placed into bits 0:4 of GPR[RT]. Bits 5:63 of GPR[RT] are set to 0.

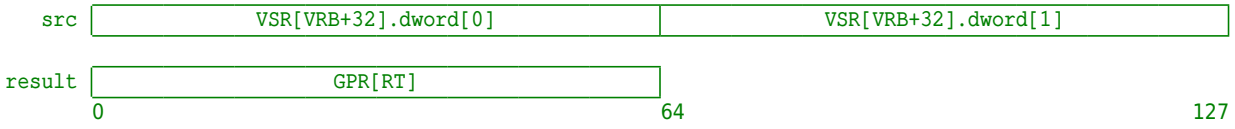
Special Registers Altered:

None

Register Data Layout for vcntmbw



Register Data Layout for vcntmbd



6.13.4 Vector Extract Mask Instructions

The *Vector Extract Mask* instructions extracts bit 0 of each element from a VSR into a GPR.

Vector Extract Byte Mask VX-form

vextractbm RT,VRB

4	RT	8	VRB	1602
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 15
   GPR[RT].bit[48+i] ← VSR[VRB+32].byte[i].bit[0]
end

GPR[RT].bit[0:47] ← 0
```

The contents of bit 0 of each byte element of VSR[VRB+32] are concatenated and placed into bits 48:63 of GPR[RT]. Bits 0:47 of GPR[RT] are set to 0.

Special Registers Altered:

None

Vector Extract Halfword Mask VX-form

vextracthm RT,VRB

4	RT	9	VRB	1602
0	6	11	16	21 31

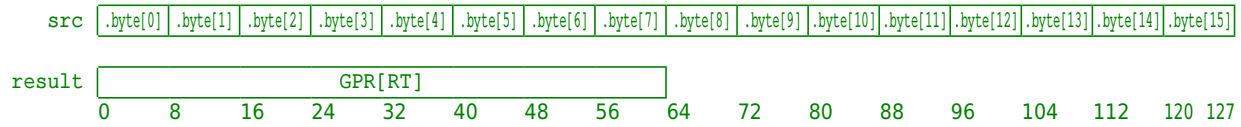
```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 7
   GPR[RT].bit[56+i] ← VSR[VRB+32].hword[i].bit[0]
end

GPR[RT].bit[0:55] ← 0
```

The contents of bit 0 of each halfword element of VSR[VRB+32] are concatenated and placed into bits 56:63 of GPR[RT]. Bits 0:55 of GPR[RT] are set to 0.

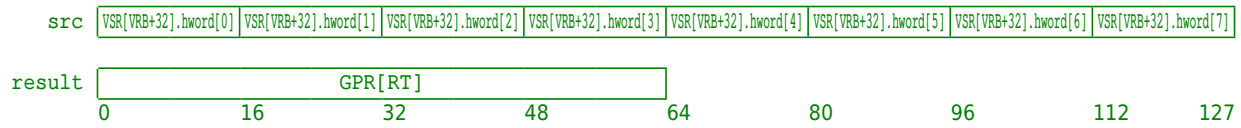
Special Registers Altered:

None

Register Data Layout for vextractbm



Register Data Layout for vextracthm



Vector Extract Word Mask VX-form

vextractwm RT,VRB

4	RT	10	VRB	1602
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
   GPR[RT].bit[60+i] ← VSR[VRB+32].word[i].bit[0]
end

GPR[RT].bit[0:59] ← 0
```

The contents of bit 0 of each word element of VSR[VRB+32] are concatenated and placed into bits 60:63 of GPR[RT]. Bits 0:59 of GPR[RT] are set to 0.

Special Registers Altered:

None

Vector Extract Doubleword Mask VX-form

vextractdm RT,VRB

4	RT	11	VRB	1602
0	6	11	16	21 31

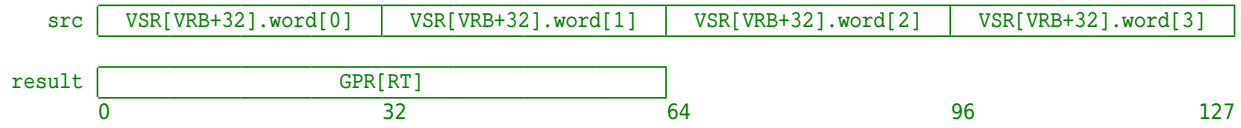
```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 1
   GPR[RT].bit[62+i] ← VSR[VRB+32].dword[i].bit[0]
end

GPR[RT].bit[0:61] ← 0
```

The contents of bit 0 of each doubleword element of VSR[VRB+32] are concatenated and placed into bits 62:63 of GPR[RT]. Bits 0:61 of GPR[RT] are set to 0.

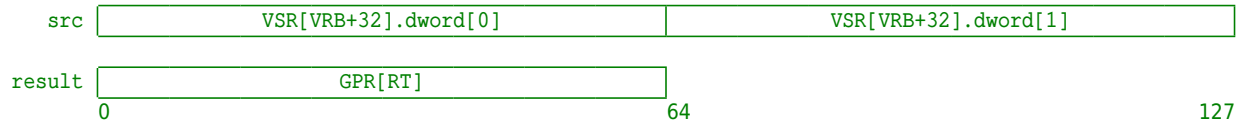
Special Registers Altered:

None

Register Data Layout for vextractwm



Register Data Layout for vextractdm



Vector Extract Quadword Mask VX-form

vextractqm RT,VRB

4	RT	12	VRB	1602
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

GPR[RT] ← EXTZ64(VSR[VRB+32].bit[0])
```

The contents of bit 0 of `VSR[VRB+32]` are placed into bit 63 of `GPR[RT]`. Bits 0:62 of `GPR[RT]` are set to 0.

Special Registers Altered:

None

Register Data Layout for vextractqm

6.14 Vector String Instructions

6.14.1 Vector String Isolate Instructions

Vector String Isolate Byte Right-justified VC-form

vstribr	VRT,VRB	(Rc=0)
vstribr.	VRT,VRB	(Rc=1)

4	VRT	1	VRB	Rc	13
0	6	11	16	21	22 31

```
if MSR.VEC=0 then Vector_Unavailable()

null_found ← 0

while (!null_found) do i = 0 to 15
   null_found ← (VSR[VRB+32].byte[15-i] = 0)
   VSR[VRT+32].byte[15-i] ← VSR[VRB+32].byte[15-i]
end

do j = i to 15
   VSR[VRT+32].byte[15-j] ← 0
end

if Rc=1 then
   CR.field[6] ← 0b00 || null_found || 0b0
```

From right to left, the contents of each byte element of VSR[VRB+32] are placed into the corresponding byte element in VSR[VRT+32]. If a byte element in VSR[VRB+32] is found to contain 0, the corresponding byte element and all byte elements to the left of that byte element in VSR[VRT+32] are set to 0.

Special Registers Altered:

Register	*Field(s)*	
CR	CR6	(if Rc=1)

Vector String Isolate Byte Left-justified VC-form

vstribl	VRT,VRB	(Rc=0)
vstribl.	VRT,VRB	(Rc=1)

4	VRT	0	VRB	Rc	13
0	6	11	16	21	22 31

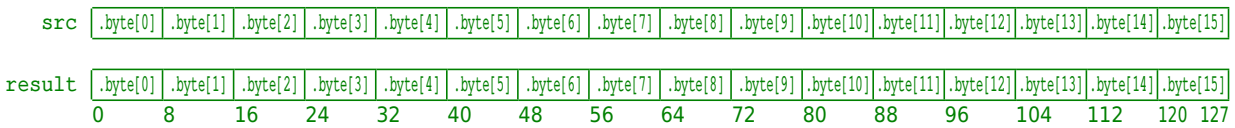
```
if MSR.VEC=0 then Vector_Unavailable()

null_found ← 0

while (!null_found) do i = 0 to 15
   null_found ← (VSR[VRB+32].byte[i] = 0)
   VSR[VRT+32].byte[i] ← VSR[VRB+32].byte[i]
end

do j = i to 15
   VSR[VRT+32].byte[j] ← 0
end

if Rc=1 then
   CR.field[6] ← 0b00 || null_found || 0b0
```

From left to right, the contents of each byte element of VSR[VRB+32] are placed into the corresponding byte element in VSR[VRT+32]. If a byte element in VSR[VRB+32] is found to contain 0, the corresponding byte element and all byte elements to the right of that byte element in VSR[VRT+32] are set to 0.

Special Registers Altered:

Register	*Field(s)*	
CR	CR6	(if Rc=1)

Register Data Layout for vstribr[.] & vstribl[.]



Vector String Isolate Halfword Right-justified VC-form

vstrihr	VRT,VRB	(Rc=0)
vstrihr.	VRT,VRB	(Rc=1)

4	VRT	3	VRB	Rc	13
0	6	11	16	21	22 31

```
if MSR.VEC=0 then Vector_Unavailable()

null_found ← 0

while (!null_found) do i = 0 to 7
   null_found ← (VSR[VRB+32].hword[7-i] = 0)
   VSR[VRT+32].hword[7-i] ← VSR[VRB+32].hword[7-i]
end

do j = i to 7
   VSR[VRT+32].hword[7-j] ← 0
end

if Rc=1 then
   CR.field[6] ← 0b00 || null_found || 0b0
```

From right to left, the contents of each halfword element of VSR[VRB+32] are placed into the corresponding halfword element in VSR[VRT+32]. If a halfword element in VSR[VRB+32] is found to contain 0, the corresponding halfword element and all halfword elements to the left of that halfword element in VSR[VRT+32] are set to 0.

Special Registers Altered:

Register	*Field(s)*	
CR	CR6	(if Rc=1)

Vector String Isolate Halfword Left-justified VC-form

vstrihl	VRT,VRB	(Rc=0)
vstrihl.	VRT,VRB	(Rc=1)

4	VRT	2	VRB	Rc	13
0	6	11	16	21	22 31

```
if MSR.VEC=0 then Vector_Unavailable()

null_found ← 0

while (!null_found) do i = 0 to 7
   null_found ← (VSR[VRB+32].hword[i] = 0)
   VSR[VRT+32].hword[i] ← VSR[VRB+32].hword[i]
end

do j = i to 7
   VSR[VRT+32].hword[j] ← 0
end

if Rc=1 then
   CR.field[6] ← 0b00 || null_found || 0b0
```

From left to right, the contents of each halfword element of VSR[VRB+32] are placed into the corresponding halfword element in VSR[VRT+32]. If a halfword element in VSR[VRB+32] is found to contain 0, the corresponding halfword element and all halfword elements to the right of that halfword element in VSR[VRT+32] are set to 0.

Special Registers Altered:

Register	*Field(s)*	
CR	CR6	(if Rc=1)

Register Data Layout for vstrihr[.] & vstrihl[.]

src	VSR[VRB+32].hword[0]	VSR[VRB+32].hword[1]	VSR[VRB+32].hword[2]	VSR[VRB+32].hword[3]	VSR[VRB+32].hword[4]	VSR[VRB+32].hword[5]	VSR[VRB+32].hword[6]	VSR[VRB+32].hword[7]
result	VSR[VRT+32].hword[0]	VSR[VRT+32].hword[1]	VSR[VRT+32].hword[2]	VSR[VRT+32].hword[3]	VSR[VRT+32].hword[4]	VSR[VRT+32].hword[5]	VSR[VRT+32].hword[6]	VSR[VRT+32].hword[7]
	0	16	32	48	64	80	96	112 127

6.14.2 Vector Clear Bytes Instructions

Vector Clear Leftmost Bytes VX-form

vclrlb VRT,VRA,RB

4	VRT	VRA	RB	397
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

N ← (GPR[RB] > 15) ? 16: GPR[RB]

do i = 0 to N-1
   VSR[VRT+32].byte[15-i] ← VSR[VRA+32].byte[15-i]
end

do i = N to 15
   VSR[VRT+32].byte[15-i] ← 0x00
end
```

Let N be the integer value in GPR[RB], or the integer value 16 if the integer value in GPR[RB] is greater than 15.

The contents of VSR[VRA+32] are placed into VSR[VRT+32] with the leftmost 16–N bytes of VSR[VRT+32] set to 0.

Special Registers Altered:

None

Vector Clear Rightmost Bytes VX-form

vclrrb VRT,VRA,RB

4	VRT	VRA	RB	461
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

N ← (GPR[RB] > 15) ? 16: GPR[RB]

do i = 0 to N-1
   VSR[VRT+32].byte[i] ← VSR[VRA+32].byte[i]
end

do i = N to 15
   VSR[VRT+32].byte[i] ← 0x00
end
```

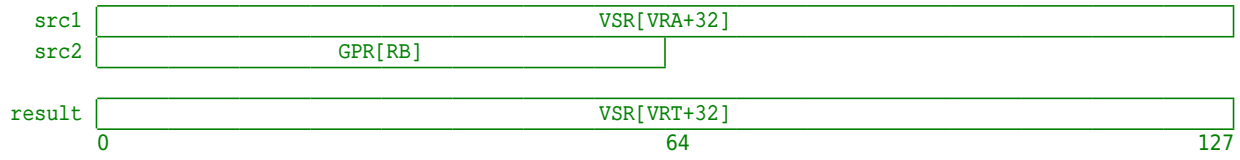
Let N be the integer value in GPR[RB], or the integer value 16 if the integer value in GPR[RB] is greater than 15.

The contents of VSR[VRA+32] are placed into VSR[VRT+32] with the rightmost 16–N bytes of VSR[VRT+32] set to 0.

Special Registers Altered:

None

Register Data Layout for vclrlb & vclrrb



6.15 Decimal Integer Instructions

A *valid encoding* of a packed decimal integer value requires the following properties.

- Each of the 31 4-bit digits of the operand's magnitude (bits 0:123) must be in the range 0-9.
- The sign code (bits 124:127) must be in the range 10-15.

Source operands with sign codes of `0b1010`, `0b1100`, `0b1110`, and `0b1111` are interpreted as positive values.

Source operands with sign codes of `0b1011` and `0b1101` are interpreted as negative values.

Positive and zero results are encoded with a either sign code of `0b1100` or `0b1111`, depending on the preferred sign (indicated as an immediate operand).

Negative results are encoded with a sign code of `0b1101`.

6.15.1 Decimal Integer Arithmetic Instructions

The *Decimal Integer Arithmetic* instructions operate on decimal integer values only in signed packed decimal format. Signed packed decimal format consists of 31 4-bit base-10 digits of magnitude and a trailing 4-bit sign code. Operations are performed as sign-magnitude, and produce a decimal result placed in a VSR (i.e., ***bcdadd.***, ***bcdsub.***).

Decimal Add Modulo VX-form

bcdadd. VRT,VRA,VRB,PS

4	VRT	VRA	VRB	1	PS	1
0	6	11	16	21	22	23 31

```
if MSR.VEC=0 then Vector_Unavailable()

VSR[VRT+32] ← bcd_ADD(VSR[VRA+32],VSR[VRB+32],PS)

CR.bit[56] ← inv_flag ? 0b0 : lt_flag
CR.bit[57] ← inv_flag ? 0b0 : gt_flag
CR.bit[58] ← inv_flag ? 0b0 : eq_flag
CR.bit[59] ← ox_flag | inv_flag
```

Let src1 be the decimal integer value in VSR[VRA+32].
Let src2 be the decimal integer value in VSR[VRB+32].

src1 is added to src2.

If the unbounded result is equal to zero, do the following.

If PS=0, the sign code of the result is set to 0b1100.
If PS=1, the sign code of the result is set to 0b1111.

CR field 6 is set to 0b0010.

If the unbounded result is greater than zero, do the following.

If PS=0, the sign code of the result is set to 0b1100.
If PS=1, the sign code of the result is set to 0b1111.

If the operation overflows, CR field 6 is set to 0b0101.
Otherwise, CR field 6 is set to 0b0100.

If the unbounded result is less than zero, do the following.

The sign code of the result is set to 0b1101.

If the operation overflows, CR field 6 is set to 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 VSR[VRT+32].

The sign code is placed in bits 124:127 of VSR[VRT+32].

If either src1 or src2 is an *invalid encoding* of a 31-digit signed decimal value, the result is undefined and CR field 6 is set to 0b0001.

Special Registers Altered:

Register	*Field(s)*
CR	CR6

Decimal Subtract Modulo VX-form

bcdsub. VRT,VRA,VRB,PS

4	VRT	VRA	VRB	1	PS	65
0	6	11	16	21	22	23 31

```
if MSR.VEC=0 then Vector_Unavailable()

VSR[VRT+32] ← bcd_SUBTRACT(VSR[VRA+32],VSR[VRB+32],PS)

CR.bit[56] ← inv_flag ? 0b0 : lt_flag
CR.bit[57] ← inv_flag ? 0b0 : gt_flag
CR.bit[58] ← inv_flag ? 0b0 : eq_flag
CR.bit[59] ← ox_flag | inv_flag
```

Let src1 be the decimal integer value in VSR[VRA+32].
Let src2 be the decimal integer value in VSR[VRB+32].

src1 is subtracted by src2.

If the unbounded result is equal to zero, do the following.

If PS=0, the sign code of the result is set to 0b1100.
If PS=1, the sign code of the result is set to 0b1111.

CR field 6 is set to 0b0010.

If the unbounded result is greater than zero, do the following.

If PS=0, the sign code of the result is set to 0b1100.
If PS=1, the sign code of the result is set to 0b1111.

If the operation overflows, CR field 6 is set to 0b0101.
Otherwise, CR field 6 is set to 0b0100.

If the unbounded result is less than zero, do the following.

The sign code of the result is set to 0b1101.

If the operation overflows, CR field 6 is set to 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 VSR[VRT+32].

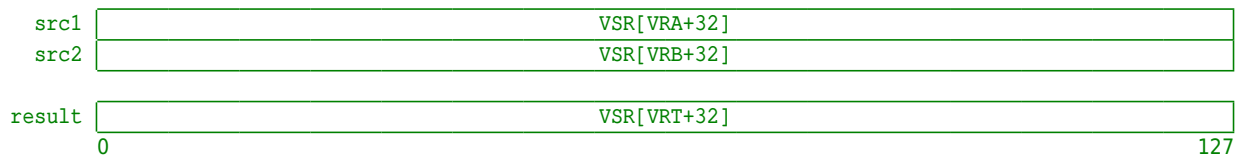
The sign code is placed in bits 124:127 of VSR[VRT+32].

If either src1 or src2 is an *invalid encoding* of a 31-digit signed decimal value, the result is undefined and CR field 6 is set to 0b0001.

Special Registers Altered:

Register	*Field(s)*
CR	CR6

Register Data Layout for bcdadd. & bcdsub.



Programming Note

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 ***bcdadd.*** and ***bcdsub.*** is a single 128-bit VSR. The register format defined for 31-digit signed decimal values employed by the *Decimal Floating-Point* instructions ***ddedpdq***[**.**] and ***denbcdq***[**.**] is a pair of 64-bit FPRs. ***xxpermdi*** can be used to convert between the two register formats as well as move data beween the FPR and VSR halves of the Vector-Scalar Registers.

fmrgew and ***fmrgow*** are provided to support direct move operations in 32-bit mode.

Programming Note

`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`).

`bcdsub. vTmp,vA,vA,0` can be used to test if an operand `vA` is an *invalid encoding* of a decimal value.

Programming Note

When bit 3 of `CR` field 6 is set to 1 by ***bcdadd.*** or ***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 ***bcdadd.*** caused bit 3 of `CR` field 6 to be set to 1, performing ***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 ***bcdsub.***, then the ***bcdadd.*** can be asserted to have overflowed. Likewise, ***bcdadd***. can be used in a similar manner to determine the cause of bit 3 of `CR` field 6 getting set by a ***bcdsub.***.

6.15.2 Decimal Integer Format Conversion Instructions

Decimal Convert From National VX-form

bcdcfn. VRT,VRB,PS

4	VRT	7	VRB	1	PS	385
0	6	11	16	21	22	23 31

```
if MSR.VEC=0 then Vector_Unavailable()

src_sign ← (VSR[VRB+32].hword[7] = 0x002D)
eq_flag  ← 1
/* check for valid sign */
inv_flag ← (VSR[VRB+32].hword[7] != 0x002B) &
           (VSR[VRB+32].hword[7] != 0x002D)

do i = 0 to 6
   eq_flag  ← eq_flag &
      (VSR[VRB+32].hword[i] = 0x0030)
   /* check for valid digit */
   inv_flag ← inv_flag |
      (VSR[VRB+32].hword[i] < 0x0030) |
      (VSR[VRB+32].hword[i] > 0x0039)
end

lt_flag ← (eq_flag=0) & (src_sign=1)
gt_flag ← (eq_flag=0) & (src_sign=0)

do i = 0 to 23
   result.nibble[i] ← 0x0
end
do i = 0 to 6
   result.nibble[i+24] ←
      VSR[VRB+32].hword[i].nibble[3]
end
result.nibble[31] ←
   (src_sign=0) ? ((PS=0) ? 0xC : 0xF) : 0xD

VSR[VRT+32] ← inv_flag ? undefined : result

CR.bit[56] ← inv_flag ? 0b0 : lt_flag
CR.bit[57] ← inv_flag ? 0b0 : gt_flag
CR.bit[58] ← inv_flag ? 0b0 : eq_flag
CR.bit[59] ← inv_flag
```

Let src be the national decimal value in VSR[VRB+32].

src is placed in VSR[VRT+32] in packed decimal format.

A valid encoding of a national decimal value requires the following.

- The contents of halfword 7 (sign code) must be either 0x002B or 0x002D.
- The contents of halfwords 0 to 6 must be in the range 0x0030 to 0x0039.

National decimal values having a sign code of 0x002B are interpreted as positive values.

National decimal values having a sign code of 0x002D are interpreted as negative values.

For each integer value i from 0 to 23, do the following.

The contents of nibble element i of VSR[VRT+32] are set to 0x0.

For each integer value i from 0 to 6, do the following.

The contents of nibble 3 of halfword element i of src are placed into nibble element i+24 of VSR[VRT+32].

For PS=0, the contents of nibble element 31 (i.e., sign code) of VSR[VRT+32] are set to 0xC for positive values and to 0xD for negative values.

For PS=1, the contents of nibble element 31 (i.e., sign code) of VSR[VRT+32] are set to 0xF for positive values and to 0xD for negative values.

CR field 6 is set to reflect src compared to zero.

If src is an *invalid encoding* of a national decimal value, the contents of VSR[VRT+32] are undefined and CR field 6 is set to 0b0001.

Special Registers Altered:

Register	*Field(s)*
CR	CR6

Register Data Layout for bcdcfn.

Decimal Convert From Zoned VX-form

bcdcfz. VRT,VRB,PS

4	VRT	6	VRB	1	PS	385
0	6	11	16	21	22	23 31

```
if MSR.VEC=0 then Vector_Unavailable()

/* check for valid sign */
inv_flag ←
   ((VSR[VRB+32].byte[15].nibble[0] < 0xA) & (PS=1)) |
    (VSR[VRB+32].byte[15].nibble[1] > 0x9)

/* check for valid digits */
MIN ← (PS=0) ? 0x30 : 0xF0
MAX ← (PS=0) ? 0x39 : 0xF9
do i = 0 to 14
   inv_flag ← inv_flag | (VSR[VRB+32].byte[i] < MIN)
                       | (VSR[VRB+32].byte[i] > MAX)
end

if PS=0 then
   src_sign ←  VSR[VRB+32].nibble[30].bit[1]
else
   src_sign ← (VSR[VRB+32].nibble[30] = 0b1011) |
              (VSR[VRB+32].nibble[30] = 0b1101)
eq_flag  ← 1

do i = 0 to 14
   result.nibble[i] ← 0x0
end
do i = 0 to 15
   result.nibble[i+15] ← VSR[VRB+32].byte[i].nibble[1]
   eq_flag ← eq_flag &
      (VSR[VRB+32].byte[i].nibble[1]=0x0)
end
lt_flag ← (eq_flag=0) & (src_sign=1)
gt_flag ← (eq_flag=0) & (src_sign=0)

result.nibble[31] ← (src_sign=0) ? 0xC : 0xD

VSR[VRT+32] ← inv_flag ? undefined : result

CR.bit[56] ← inv_flag ? 0b0 : lt_flag
CR.bit[57] ← inv_flag ? 0b0 : gt_flag
CR.bit[58] ← inv_flag ? 0b0 : eq_flag
CR.bit[59] ← inv_flag
```

Let src be the zoned decimal value in VSR[VRB+32].

src is placed in VSR[VRT+32] 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 VSR[VRT+32] are set to 0x0.

For each integer value i from 0 to 15,

The contents of nibble 1 of byte element i of src are placed into nibble element i+15 of VSR[VRT+32].

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 VSR[VRT+32] are undefined and CR field 6 is set to 0b0001.

Special Registers Altered:

Register	*Field(s)*
CR	CR6

Register Data Layout for bcdcfz.

Decimal Convert To National VX-form

bcdctn. VRT,VRB

4	VRT	5	VRB	1	/	385
0	6	11	16	21	22	23 31

```
if MSR.VEC=0 then Vector_Unavailable()

ox_flag ← 0
do i = 0 to 23
   ox_flag ← ox_flag | (VSR[VRB+32].nibble[i] != 0x0)
end

inv_flag ← (VSR[VRB+32].nibble[31] < 0xA)
do i = 0 to 30
   inv_flag ← inv_flag | (VSR[VRB+32].nibble[i] > 0x9)
end

src_sign ← (VSR[VRB+32].nibble[31] = 0xB) |
           (VSR[VRB+32].nibble[31] = 0xD)

eq_flag ← (VSR[VRB+32].nibble[0:30] = 0)
lt_flag ← (eq_flag=0) & (src_sign=1)
gt_flag ← (eq_flag=0) & (src_sign=0)

do i = 0 to 6
   result.hword[i].nibble[0:2] ← 0x003
   result.hword[i].nibble[3] ← VSR[VRB+32].nibble[i+24]
end

result.hword[7] ← (src_sign=1) ? 0x002D : 0x002B

VSR[VRT+32] ← inv_flag ? undefined : result

CR.bit[56] ← inv_flag ? 0b0 : lt_flag
CR.bit[57] ← inv_flag ? 0b0 : gt_flag
CR.bit[58] ← inv_flag ? 0b0 : eq_flag
CR.bit[59] ← inv_flag | ox_flag
```

Let src be the packed decimal value in VSR[VRB+32].

src is placed into VSR[VRT+32] in national decimal format.

A valid encoding of a signed packed decimal value requires the following.

- The contents of nibble 31 (sign code) must be a value in the range 0xA to 0xF.
- The contents of each nibble 0-30 must be a value in the range 0x0 to 0x9.

Packed decimal values with sign codes of 0xA, 0xC, 0xE, or 0xF are interpreted as positive values.

Packed decimal values with sign codes of 0xB or 0xD are interpreted as negative values.

Values greater in magnitude than 10^7 - 1 are too large to be represented in national decimal format.

For each integer value i from 0 to 6, do the following.

The value 0x003 is placed into nibbles 0:2 of halfword element i of VSR[VRT+32].

The contents of nibble element i+24 of VSR[VRB+32] are placed into nibble 3 of halfword element i of VSR[VRT+32].

The contents of halfword element 7 (i.e., sign code) of VSR[VRT+32] are set to 0x002B for positive values and to 0x002D for negative values.

CR field 6 is set to reflect src compared to zero, including whether or not src is too large to be represented in national decimal format.

If src is an *invalid encoding* of a packed decimal value, the contents of VSR[VRT+32] are undefined and CR field 6 is set to 0b0001.

Special Registers Altered:

Register	*Field(s)*
CR	CR6

Register Data Layout for bcdctn.

Decimal Convert To Zoned VX-form

bcdctz. VRT,VRB,PS

4	VRT	4	VRB	1	PS	385
0	6	11	16	21	22	23 31

```
if MSR.VEC=0 then Vector_Unavailable()

inv_flag ← (VSR[VRB+32].nibble[31] < 0xA)
do i = 0 to 30
   inv_flag ← inv_flag | (VSR[VRB+32].nibble[i] > 0x9)
end

ox_flag ← 0
do i = 0 to 15
   ox_flag ← ox_flag | (VSR[VRB+32].nibble[i] != 0x0)
end

src_sign ← (VSR[VRB+32].nibble[31] = 0xB) |
           (VSR[VRB+32].nibble[31] = 0xD)

eq_flag ← (VSR[VRB+32].nibble[0:30] = 0)
lt_flag ← (eq_flag=0) & (src_sign=1)
gt_flag ← (eq_flag=0) & (src_sign=0)

do i = 0 to 14
   result.byte[i].nibble[0] ← (PS=0) ? 0x3 : 0xF
   result.byte[i].nibble[1] ← VSR[VRB+32].nibble[i+15]
end
if src.sign=0 then
   result.byte[15].nibble[0] ← (PS=0) ? 0x3 : 0xC
else
   result.byte[15].nibble[0] ← (PS=0) ? 0x7 : 0xD

result.byte[15].nibble[1] ← VSR[VRB+32].nibble[30]

VSR[VRT+32] ← inv_flag ? undefined : result

CR.bit[56] ← inv_flag ? 0b0 : lt_flag
CR.bit[57] ← inv_flag ? 0b0 : gt_flag
CR.bit[58] ← inv_flag ? 0b0 : eq_flag
CR.bit[59] ← inv_flag | ox_flag
```

Let src be the packed decimal value in VSR[VRB+32].

src is placed into VSR[VRT+32] in zoned decimal format.

A valid encoding of a signed packed decimal value requires the following.

- The contents of nibble 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 i from 0 to 15, do the following.

The rightmost nibble of each digit i of the zoned decimal result is set to the contents of nibble i+15 of src.

The result is placed into VSR[VRT+32].

CR field 6 is set to reflect src compared to zero, including whether or not src is too large to be represented in zoned decimal format.

If src is an *invalid encoding* of a packed decimal value, the contents of VSR[VRT+32] are undefined and CR field 6 is set to 0b0001.

Special Registers Altered:

Register	*Field(s)*
CR	CR6

Register Data Layout for bcdctz.

Decimal Convert From Signed Quadword VX-form

bcdcfsq. VRT,VRB,PS

4	VRT	2	VRB	1	PS	385
0	6	11	16	21	22	23 31

```
if MSR.VEC=0 then Vector_Unavailable()

ox_flag ← (EXTS(VSR[VRB+32]) > 10^31-1) |
          (EXTS(VSR[VRB+32]) < -(10^31-1))
lt_flag ← (EXTS(VSR[VRB+32]) < 0)
gt_flag ← (EXTS(VSR[VRB+32]) > 0)
eq_flag ← (EXTS(VSR[VRB+32]) = 0)

if ox_flag=0 then
   result ← bcd_CONVERT_FROM_SI128(EXTS(VSR[VRB+32]),PS)
else
   result ← 0xUUUU_UUUU_UUUU_UUUU_UUUU_UUUU_UUUU_UUUU

VSR[VRT+32] ← ox_flag ? undefined : result

CR.bit[56] ← lt_flag
CR.bit[57] ← gt_flag
CR.bit[58] ← eq_flag
CR.bit[59] ← ox_flag
```

Let src be the signed integer value in VSR[VRB+32].

src is placed into VSR[VRT+32] in signed packed decimal format.

For PS=0, the contents of nibble element 31 (i.e., sign code) of VSR[VRT+32] 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 VSR[VRT+32] are set to 0xF for values greater than or equal to 0 and to 0xD for values less than 0.

If the signed integer value in VSR[VRB+32] is greater than $10^{31}-1$ or less than $-(10^{31}-1)$, the value is too large to be represented in packed decimal format, and the contents of VSR[VRT+32] 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:

Register	*Field(s)*
CR	CR6

Register Data Layout for bcdcfsq.

Decimal Convert To Signed Quadword VX-form

bcdctsq. VRT,VRB

4	VRT	0	VRB	1	/	385
0	6	11	16	21	22	23 31

```
if MSR.VEC=0 then Vector_Unavailable()

inv_flag ← (VSR[VRB+32].nibble[31] < 0xA)
do i = 0 to 30
   inv_flag ← inv_flag | (VSR[VRB+32].nibble[i] > 0x9)
end

src_sign    ← (VSR[VRB+32].nibble[31] = 0xB) |
              (VSR[VRB+32].nibble[31] = 0xD)

eq_flag     ← (VSR[VRB+32].nibble[0:30] = 0)
lt_flag     ← (eq_flag=0) & (src_sign=1)
gt_flag     ← (eq_flag=0) & (src_sign=0)

result      ← si128_CONVERT_FROM_BCD(VSR[VRB+32])

VSR[VRT+32] ← inv_flag ? undefined : result

CR.bit[56]  ← inv_flag ? 0b0 : lt_flag
CR.bit[57]  ← inv_flag ? 0b0 : gt_flag
CR.bit[58]  ← inv_flag ? 0b0 : eq_flag
CR.bit[59]  ← inv_flag
```

Let src be the packed decimal value in VSR[VRB+32].

src is placed into VSR[VRT+32] in signed integer format.

A valid encoding of a signed packed decimal value requires the following.

- The contents of nibble 31 (sign code) must be a value in the range 0xA to 0xF.
- The contents of each nibble, 0-30, must be a value in the range 0x0 to 0x9.

Packed decimal values with sign codes of 0xA, 0xC, 0xE, or 0xF are interpreted as positive values.

Packed decimal values with sign codes of 0xB or 0xD are interpreted as negative values.

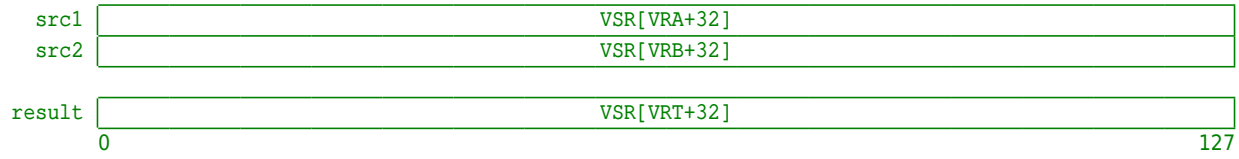
CR field 6 is set to reflect src compared to zero.

If src is an *invalid encoding* of a packed decimal value, the contents of VSR[VRT+32] are undefined and CR field 6 is set to 0b0001.

Special Registers Altered:

Register	*Field(s)*
CR	CR6

Register Data Layout for bcdctsq.



Vector Multiply-by-10 Unsigned Quadword VX-form

vmul10uq VRT,VRA

4	VRT	VRA	///	513
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

src  ← EXTZ(VSR[VRA+32])
prod ← (src << 3) + (src << 1)
VSR[VRT+32] ← CHOP128(prod)
```

Let src be the unsigned integer value in VSR[VRA+32].

The rightmost 128 bits of the product of src multiplied by the value 10 are placed into VSR[VRT+32].

Special Registers Altered:

None

Vector Multiply-by-10 & write Carry-out Unsigned Quadword VX-form

vmul10cuq VRT,VRA

4	VRT	VRA	///	1
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

src  ← EXTZ(VSR[VRA+32])
prod ← (src << 3) + (src << 1)
VSR[VRT+32] ← CHOP128(prod >> 128)
```

Let src be the unsigned integer value in VSR[VRA+32].

The product of src multiplied by the value 10 is shifted right by 128 bits. The rightmost 128 bits of the shifted result is placed into VSR[VRT+32].

Special Registers Altered:

None

Register Data Layout for vmul10uq & vmul10cuq

Vector Multiply-by-10 Extended Unsigned Quadword VX-form

vmul10euq VRT,VRA,VRB

4	VRT	VRA	VRB	577
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

src  ← EXTZ(VSR[VRA+32])
cin  ← EXTZ(VSR[VRB+32].bit[124:127])
prod ← (src << 3) + (src << 1) + cin
VSR[VRT+32] ← CHOP128(prod)
```

Let src be the unsigned integer value in VSR[VRA+32].

Let cin be the unsigned packed decimal value in bits 124:127 of VSR[VRB+32]. Values of cin greater than 9 are undefined.

The rightmost 128 bits of the sum of cin and the product of src multiplied by the value 10 are placed into VSR[VRT+32].

Special Registers Altered:

None

Vector Multiply-by-10 Extended & write Carry-out Unsigned Quadword VX-form

vmul10ecuq VRT,VRA,VRB

4	VRT	VRA	VRB	65
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

src  ← EXTZ(VSR[VRA+32])
cin  ← EXTZ(VSR[VRB+32].bit[124:127])
prod ← (src << 3) + (src << 1) + cin
VSR[VRT+32] ← CHOP128(prod>>128)
```

Let src be the unsigned integer value in VSR[VRA+32].

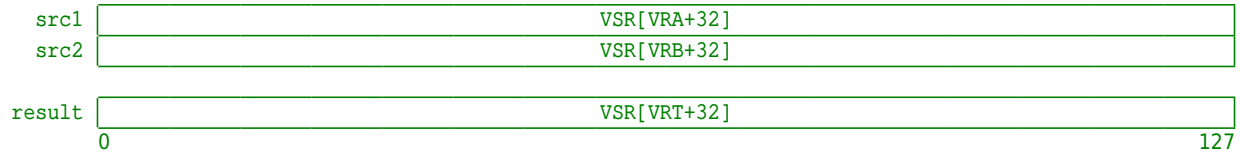
Let cin be the unsigned packed decimal value in bits 124:127 of VSR[VRA+32]. Values of cin greater than 9 are undefined.

The sum of cin and the product of src multiplied by the value 10 is shifted right by 128 bits. The rightmost 128 bits of the shifted result is placed into VSR[VRT+32].

Special Registers Altered:

None

Register Data Layout for vmul10euq & vmul10ecuq



6.15.3 Decimal Integer Sign Manipulation Instructions

Decimal Copy Sign VX-form

bcdcpsgn. VRT,VRA,VRB

4	VRT	VRA	VRB	833
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

inv_flag ← (VSR[VRA+32].nibble[31] < 0xA) |
           (VSR[VRB+32].nibble[31] < 0xA)
do i = 0 to 30
    inv_flag ← inv_flag |
               (VSR[VRA+32].nibble[i] > 0x9) |
               (VSR[VRB+32].nibble[i] > 0x9)
end

src_sign ← (VSR[VRB+32].nibble[31] = 0xB) |
           (VSR[VRB+32].nibble[31] = 0xD)

eq_flag  ← (VSR[VRA+32].nibble[0:30] = 0)
lt_flag  ← (eq_flag=0) & (src_sign=1)
gt_flag  ← (eq_flag=0) & (src_sign=0)

result.nibble[0:30] ← VSR[VRA+32].nibble[0:30]
result.nibble[31]   ← VSR[VRB+32].nibble[31]

VSR[VRT+32] ← inv_flag ? undefined : result

CR.bit[56]  ← inv_flag ? 0b0 : lt_flag
CR.bit[57]  ← inv_flag ? 0b0 : gt_flag
CR.bit[58]  ← inv_flag ? 0b0 : eq_flag
CR.bit[59]  ← inv_flag
```

The decimal value in VSR[VRA+32] is placed into VSR[VRT+32] with the sign code of the decimal value in VSR[VRB+32].

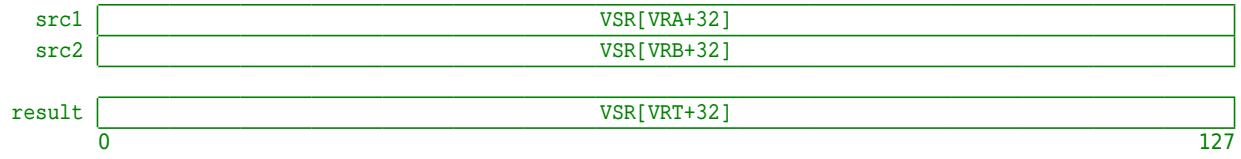
CR field 6 is set to reflect the result compared to zero.

If either the decimal value in VSR[VRA+32] or the decimal value in VSR[VRB+32] is an *invalid encoding*, the contents of VSR[VRT+32] are undefined and CR field 6 is set to 0b0001.

Special Registers Altered:

Register	*Field(s)*
CR	CR6

Register Data Layout for bcdcpsgn.



Decimal Set Sign VX-form

bcdsetsgn. VRT,VRB,PS

4	VRT	31	VRB	1	PS	385
0	6	11	16	21	22	23 31

```
if MSR.VEC=0 then Vector_Unavailable()

inv_flag ← (VSR[VRB+32].nibble[31] < 0xA)
do i = 0 to 30
    inv_flag ← inv_flag |
               (VSR[VRB+32].nibble[i] > 0x9)
end

src_sign ← (VSR[VRB+32].nibble[31] = 0xB) |
            (VSR[VRB+32].nibble[31] = 0xD)

eq_flag  ← (VSR[VRB+32].nibble[0:30] = 0)
lt_flag  ← (eq_flag=0) & (src_sign=1)
gt_flag  ← (eq_flag=0) & (src_sign=0)

result.nibble[0:30] ← VSR[VRB+32].nibble[0:30]
result.nibble[31]   ←
   (src_sign=0) ? ((PS=0) ? 0xC:0xF) : 0xD

VSR[VRT+32] ← inv_flag ? undefined : result

CR.bit[56]  ← inv_flag ? 0b0 : lt_flag
CR.bit[57]  ← inv_flag ? 0b0 : gt_flag
CR.bit[58]  ← inv_flag ? 0b0 : eq_flag
CR.bit[59]  ← inv_flag
```

Let `src` be the packed decimal value in `VSR[VRB+32]`.

Packed decimal values with sign codes of `0xA`, `0xC`, `0xE`, or `0xF` are interpreted as positive values.

Packed decimal values with sign codes of `0xB` or `0xD` are interpreted as negative values.

If `src` is negative, `src` is placed into `VSR[VRT+32]` with the sign code set to `0xD`.

If `src` is positive and `PS=0`, `src` is placed into `VSR[VRT+32]` with the sign code set to `0xC`.

If `src` is positive and `PS=1`, `src` is placed into `VSR[VRT+32]` with the sign code set to `0xF`.

CR field 6 is set to reflect `src` compared to zero.

If `src` is an *invalid encoding* of a packed decimal value, the contents of `VSR[VRT+32]` are undefined and `CR` field `6` is set to `0b0001`.

Special Registers Altered:

Register	*Field(s)*
CR	CR6

Register Data Layout for bcdsetsgn.

6.15.4 Decimal Integer Shift and Round Instructions

Decimal Shift VX-form

bcds. VRT,VRA,VRB,PS

4	VRT	VRA	VRB	1	PS	193
0	6	11	16	21	22	23 31

```
if MSR.VEC=0 then Vector_Unavailable()

n ← EXTS(VSR[VRA+32].byte[7])

inv_flag ← (VSR[VRB+32].nibble[31] < 0xA)
do i = 0 to 30
   inv_flag ← inv_flag |
              (VSR[VRB+32].nibble[i] > 0x9)
end

src_sign ← (VSR[VRB+32].nibble[31] = 0xB) |
           (VSR[VRB+32].nibble[31] = 0xD)

eq_flag  ← (VSR[VRB+32].nibble[0:30] = 0)
lt_flag  ← (eq_flag=0) & (src_sign=1)
gt_flag  ← (eq_flag=0) & (src_sign=0)

if n > 0 then do     // shift left
   shcnt               ← (n<32) ? n : 31
   src.nibble[0:30]    ← VSR[VRB+32].nibble[0:30]
   src.nibble[31:61]   ← 0
   result.nibble[0:30] ← src.data.nibble[shcnt:shcnt+30]
   ox_flag             ← (shcnt > 0) &
                         (src.nibble[0:shcnt-1] != 0)
end
else do              // shift right
   shcnt               ← ((¬n+1)<32) ? (¬n+1) : 31
   src.nibble[0:30]    ← 0
   src.nibble[31:61]   ← VSR[VRB+32].nibble[0:30]
   result.nibble[0:30] ← src.nibble[31-shcnt:61-shcnt]
   ox_flag ← 0b0
end

result.nibble[31] ←
   (src_sign=0) ? ((PS=0) ? 0xC : 0xF) : 0xD

VSR[VRT+32] ← inv_flag ? undefined : result

CR.bit[56] ← inv_flag ? 0b0 : lt_flag
CR.bit[57] ← inv_flag ? 0b0 : gt_flag
CR.bit[58] ← inv_flag ? 0b0 : eq_flag
CR.bit[59] ← inv_flag | ox_flag
```

Let n be the signed integer value in byte element 7 of VSR[VRA+32].

Let src be the signed packed decimal value in VSR[VRB+32].

A valid encoding of a signed packed decimal value requires the following.

- The contents of nibble 31 (sign code) must be a value in the range 0xA to 0xF.
- The contents of each nibble 0-30 must be a value in the range 0x0 to 0x9.

Packed decimal source operands with sign codes of 0xA, 0xC, 0xE, or 0xF are interpreted as positive values.

Packed decimal source operands with sign codes of 0xB or 0xD are interpreted as negative values.

If n is greater than zero, src is shifted left n digits. Zeros are supplied to vacated digits on the right. If any non-zero digits are shifted out, an overflow occurs.

If n is less than zero, src is shifted right −n digits. Zeros are supplied to vacated digits on the left.

If the packed decimal value in VSR[VRB+32] is negative, the sign code of the result is set to 0b1101.

If the packed decimal value in VSR[VRB+32] is positive, the sign code of the result is set to 0b1100 if PS=0 and is set to 0b1111 if PS=1.

The shifted result is placed into VSR[VRT+32].

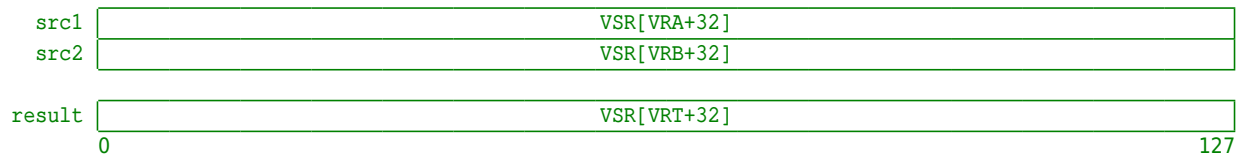
CR field 6 is set to reflect src compared to zero, including whether or not significant digits were shifted out when the shift count is positive (i.e., left shift operation).

If src is an *invalid encoding* of a packed decimal value, the contents of VSR[VRT+32] are undefined and CR field 6 is set to 0b0001.

Special Registers Altered:

Register	*Field(s)*
CR	CR6

Register Data Layout for bcds.



Decimal Unsigned Shift VX-form

bcdus. VRT,VRA,VRB

4	VRT	VRA	VRB	1	/	129
0	6	11	16	21	22	23 31

```
if MSR.VEC=0 then Vector_Unavailable()

n ← EXTS(VSR[VRA+32].byte[7])

inv_flag ← 0
do i = 0 to 31
   inv_flag ← inv_flag |
               (VSR[VRB+32].nibble[i] > 0x9)
end

eq_flag  ← (VSR[VRB+32].nibble[0:31] = 0)
gt_flag  ← (eq_flag=0)

if n > 0 then do     // shift left
   shcnt ← (n<33) ? n : 32
   src.nibble[0:31] ← VSR[VRB+32]
   src.nibble[32:63] ← 0
   result ← src.nibble[shcnt:shcnt+31]
   ox_flag ← (shcnt > 0) & (src.nibble[0:shcnt-1] !=
0)
end
else do              // shift right
   shcnt ← ((¬n+1)<33) ? (¬n+1) : 32
   src.nibble[0:31]  ← 0
   src.nibble[32:63] ← VSR[VRB+32]
   result ← src.nibble[32-shcnt:63-shcnt]
   ox_flag ← 0
end

VSR[VRT+32] ← inv_flag ? undefined : result

CR.bit[56] ← 0b0
CR.bit[57] ← inv_flag ? 0b0 : gt_flag
CR.bit[58] ← inv_flag ? 0b0 : eq_flag
CR.bit[59] ← inv_flag | ox_flag
```

Let `n` be the signed integer value in byte element `7` of `VSR[VRA+32]`.

Let `src` be the unsigned packed decimal value in `VSR[VRB+32]`.

A valid encoding of an unsigned packed decimal value requires the contents of each nibble 0-31 must be a value in the range `0x0` to `0x9`.

If `n` is greater than zero, `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 `VSR[VRT+32]`.

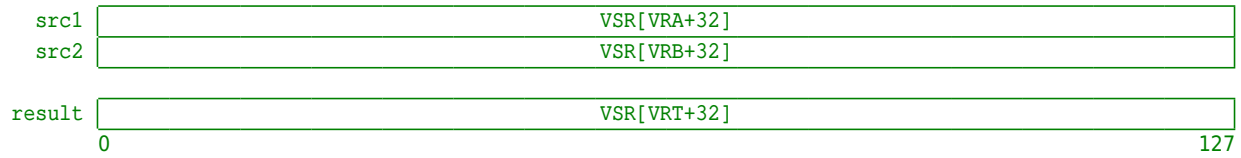
CR field 6 is set to reflect `src` compared to zero, including whether or not significant digits were shifted out when the shift count is positive (i.e., left shift operation).

If `src` is an *invalid encoding* of a packed decimal value, the contents of `VSR[VRT+32]` are undefined and `CR` field `6` is set to `0b0001`.

Special Registers Altered:

Register	*Field(s)*
CR	CR6

Register Data Layout for bcdus.



Decimal Shift & Round VX-form

bcdsr. VRT,VRA,VRB,PS

4	VRT	VRA	VRB	1	PS	449
0	6	11	16	21	22	23 31

```
if MSR.VEC=0 then Vector_Unavailable()

n ← EXTS(VSR[VRA+32].byte[7])

inv_flag ← (VSR[VRB+32].nibble[31] < 0xA)
do i = 0 to 30
   inv_flag ← inv_flag |
               (VSR[VRB+32].nibble[i] > 0x9)
end

src_sign ← (VSR[VRB+32].nibble[31] = 0xB) |
           (VSR[VRB+32].nibble[31] = 0xD)

eq_flag  ← (VSR[VRB+32].nibble[0:30] = 0)
lt_flag  ← (eq_flag=0) & (src_sign=1)
gt_flag  ← (eq_flag=0) & (src_sign=0)

if n > 0 then do     // shift left
   shcnt ← Clamp(n, 0, 31)
   src.nibble[0:30] ← VSR[VRB+32].nibble[0:30]
   src.nibble[31:61] ← 0
   result.nibble[0:30] ← src.nibble[shcnt:shcnt+30]
   ox_flag ← (shcnt > 0) &
              (src.nibble[0:shcnt-1] != 0)
   g_flag  ← 0
end
else do              // shift right
   shcnt ← Clamp(¬n + 1, 0, 31)
   src.nibble[0:30] ← 0
   src.nibble[31:61] ← VSR[VRB+32].nibble[0:30]
   result.nibble[0:30] ← src.nibble[31-shcnt:61-shcnt]
   ox_flag ← 0
   g_flag  ← (shcnt > 0) &
              (EXTZ(src.nibble[62-shcnt]) >= 5)
end
result.nibble[31] ←
   (src_sign=0) ? ((PS=0) ? 0xC : 0xF) : 0xD

result ← (g_flag=0) ? result : bcd_INCREMENT(result)

VSR[VRT+32] ← inv_flag ? undefined : result

CR.bit[56] ← inv_flag ? 0b0 : lt_flag
CR.bit[57] ← inv_flag ? 0b0 : gt_flag
CR.bit[58] ← inv_flag ? 0b0 : eq_flag
CR.bit[59] ← inv_flag | ox_flag
```

Let `n` be the signed integer value in byte element `7` of `VSR[VRA+32]`.

Let `src` be the signed packed decimal value in `VSR[VRB+32]`.

A valid encoding of a signed packed decimal source operand requires the following.

- The contents of nibble 31 (sign code) must be a value in the range `0xA` to `0xF`.
- The contents of each nibble 0-30 must be a value in the range `0x0` to `0x9`.

Packed decimal source operands with sign codes of `0xA`, `0xC`, `0xE`, or `0xF` are interpreted as positive values.

Packed decimal source operands with sign codes of `0xB` or `0xD` are interpreted as negative values.

If `n` is greater than zero, `src` is shifted left `n` digits. Zeros are supplied to vacated digits on the right. If any non-zero digits are shifted out, an overflow occurs.

If `n` is less than zero, `src` is shifted right `-n` digits. Zeros are supplied to vacated digits on the left. If the value of the last nibble shifted out on the right was greater than or equal to 5, the magnitude of the result is incremented by 1.

If `src` is negative, the sign code of the result is set to `0b1101`.

If `src` is positive, the sign code of the result is set to `0b1100` if `PS=0` and is set to `0b1111` if `PS=1`.

The shifted and rounded result is placed into `VSR[VRT+32]`.

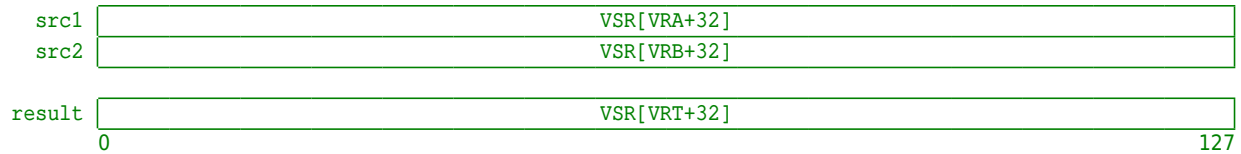
CR field 6 is set to reflect `src` compared to zero, including whether or not significant digits were shifted out when the shift count is positive (i.e., left shift operation).

If `src` is an *invalid encoding* of a packed decimal value, the contents of `VSR[VRT+32]` are undefined and `CR` field `6` is set to `0b0001`.

Special Registers Altered:

Register	*Field(s)*
CR	CR6

Register Data Layout for bcdsr.



6.15.5 Decimal Integer Truncate Instructions

Decimal Truncate VX-form

bcdtrunc. VRT,VRA,VRB,PS

4	VRT	VRA	VRB	1	PS	257
0	6	11	16	21	22	23 31

```
if MSR.VEC=0 then Vector_Unavailable()

inv_flag ← (VSR[VRB+32].nibble[31] < 0xA)
do i = 0 to 30
    inv_flag ← inv_flag |
                (VSR[VRB+32].nibble[i] > 0x9)
end

length  ← VSR[VRA+32].bit[48:63]

ox_flag ← 0

src_sign ← (VSR[VRB+32].nibble[31] = 0xB) |
           (VSR[VRB+32].nibble[31] = 0xD)

eq_flag  ← (VSR[VRB+32].nibble[0:30] = 0)
lt_flag  ←  src_sign & ¬eq_flag
gt_flag  ← ¬src_sign & ¬eq_flag

if length < 31 then do
   do i = 0 to 30-length
      if VSR[VRB+32].nibble[i]!=0b0000 then
         ox_flag ← 1
      result.nibble[i] ← 0b0000
   end
   if length > 0 then do
      do i = 31-length to 30
         result.nibble[i] ← VSR[VRB+32].nibble[i]
      end
   end
end
else
   result.nibble[0:30] ← VSR[VRB+32].nibble[0:30]

result.nibble[31] ←
   (src_sign=0) ? ((PS=0) ? 0xC : 0xF) : 0xD

VSR[VRT+32] ← inv_flag ? undefined : result

CR.bit[56] ← inv_flag ? 0b0 : lt_flag
CR.bit[57] ← inv_flag ? 0b0 : gt_flag
CR.bit[58] ← inv_flag ? 0b0 : eq_flag
CR.bit[59] ← inv_flag | ox_flag
```

Let `length` be the integer value in bits 48:63 of `VSR[VRA+32]`.

Let `src` be the signed decimal value in `VSR[VRB+32]`.

A valid encoding of a packed decimal source operand requires the following.

- The contents of nibble 31 (sign code) must be a value in the range `0xA` to `0xF`.
- The contents of each nibble 0-30 must be a value in the range `0x0` to `0x9`.

Packed decimal values with sign codes of `0xA`, `0xC`, `0xE`, or `0xF` are interpreted as positive values.

Packed decimal values with sign codes of `0xB` or `0xD` are interpreted as negative values.

If `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`.

`src` is copied into `VSR[VRT+32]` with the leftmost `31-length` digits each set to `0b0000`. If any of the leftmost `31-length` digits of the signed decimal value in `VSR[VRB+32]` are non-zero, an overflow occurs.

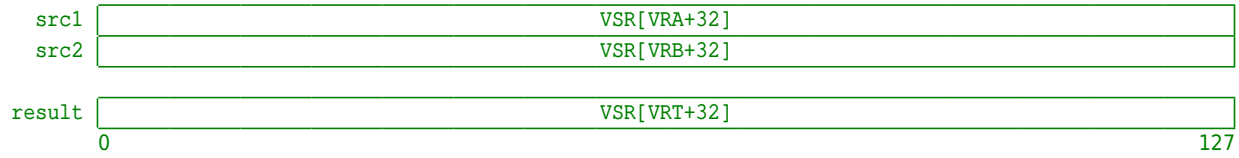
CR field 6 is set to reflect `src` compared to zero, including whether or not significant digits were truncated.

If `src` is an *invalid encoding* of a packed decimal value, the contents of `VSR[VRT+32]` are undefined and `CR` field `6` is set to `0b0001`.

Special Registers Altered:

Register	*Field(s)*
CR	CR6

Register Data Layout for bcdtrunc.



Decimal Unsigned Truncate VX-form

bcdutrunc. VRT,VRA,VRB

4	VRT	VRA	VRB	1	/	321
0	6	11	16	21	22	23 31

```
if MSR.VEC=0 then Vector_Unavailable()

inv_flag ← 0
do i = 0 to 31
    inv_flag ← inv_flag |
                (VSR[VRB+32].nibble[i] > 0x9)
end

length ← VSR[VRA+32].bit[48:63]

ox_flag ← 0

eq_flag  ← (VSR[VRB+32].nibble[0:31]  = 0)
gt_flag  ← (VSR[VRB+32].nibble[0:31] != 0)

if length < 32 then do
   do i = 0 to 31-length
      if VSR[VRB+32].nibble[i]!=0b0000 then
         ox_flag ← 1
      result.nibble[i] ← 0b0000
   end
   if length > 0 then do
      do i = 32-length to 31
         result.nibble[i] ← VSR[VRB+32].nibble[i]
      end
   end
end
else result ← VSR[VRB+32]

VSR[VRT+32] ← inv_flag ? undefined : result

CR.bit[56] ← 0b0
CR.bit[57] ← inv_flag ? 0b0 : gt_flag
CR.bit[58] ← inv_flag ? 0b0 : eq_flag
CR.bit[59] ← inv_flag | ox_flag
```

Let `length` be the integer value in bits 48:63 of `VSR[VRA+32]`.

Let `src` be the unsigned decimal value in `VSR[VRB+32]`.

A valid encoding of a packed decimal source operand requires the contents of each nibble 0-31 must be a value in the range `0x0` to `0x9`.

`src` is copied into `VSR[VRT+32]` with the leftmost `32-length` digits each set to `0b0000`. If any of the leftmost `32-length` digits of the signed decimal value in `VSR[VRB+32]` are non-zero, an overflow occurs.

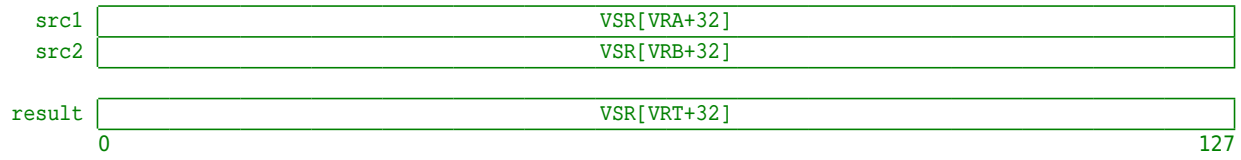
`CR` field 6 is set to reflect `src` compared to zero, including whether or not significant digits were truncated.

If `src` is an *invalid encoding* of a packed decimal value, the contents of `VSR[VRT+32]` are undefined and `CR` field `6` is set to `0b0001`.

Special Registers Altered:

Register	*Field(s)*
CR	CR6

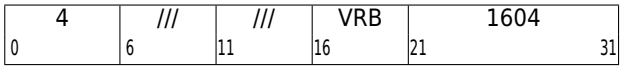
Register Data Layout for bcdutrunc.



6.16 Vector Status and Control Register Instructions

Move To Vector Status and Control Register VX-form

mtvscr VRB




```
if MSR.VEC=0 then Vector_Unavailable()

VSCR ← VSR[VRB+32].word[3]
```

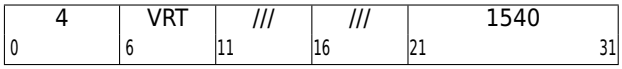
The contents of word element 3 of `VSR[VRB+32]` are placed into the VSCR.

Special Registers Altered:

None

Move From Vector Status and Control Register VX-form

mfvscr VRT




```
if MSR.VEC=0 then Vector_Unavailable()

VSR[VRT+32] ← EXTZ128(VSCR)
```

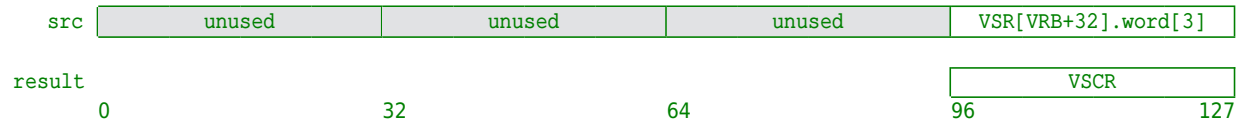
The contents of the VSCR are placed into word element 3 of `VSR[VRT+32]`.

The remaining word elements in `VSR[VRT+32]` are set to 0.

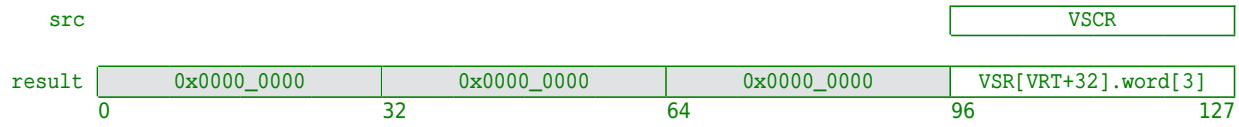
Special Registers Altered:

None

Register Data Layout for mtvscr



Register Data Layout for mfvscr



Chapter 7. Vector-Scalar Extension Facility

7.1 Introduction

7.1.1 Overview of the Vector-Scalar Extension

Vector-Scalar Extension (VSX) provides facilities supporting vector and scalar binary floating-point operations. The following VSX features are provided to increase opportunities for vectorization.

- A unified register file, a set of Vector-Scalar Registers (VSR), supporting both scalar and vector operations is provided, eliminating the overhead of vector-scalar data transfer through storage.
- Support for word-aligned storage accesses for both scalar and vector operations is provided.
- Robust support for IEEE-754 for both vector and scalar floating-point operations is provided.

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.

Programming Note

An application binary interface defined to support Vector-Scalar operations should also specify a requirement that Floating-Point operations and Vector operations are made available to the application when Vector-Scalar operations are made available.

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 set to 0.

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 7.1 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 7.2. Many instructions perform a given operation in parallel on all elements in a VSR. Depending on the instruction, a word element can be interpreted as a signed integer word (SW), an unsigned integer word (UW), a logical mask value (MW), or a single-precision floating-point value (SP); a doubleword element can be interpreted as a doubleword signed integer (SD), a doubleword unsigned integer (UD), a doubleword mask (DM), or a double-precision floating-point value (DP). In the instructions descriptions, phrases like *signed integer word element* are used as shorthand for *word element, interpreted as a signed integer*.

Load and *Store* instructions are provided that transfer a byte, a specified number of bytes (up to 16), a halfword, a word, a doubleword, or a quadword between storage and a VSR, or an octword between storage and a pair of VSRs.

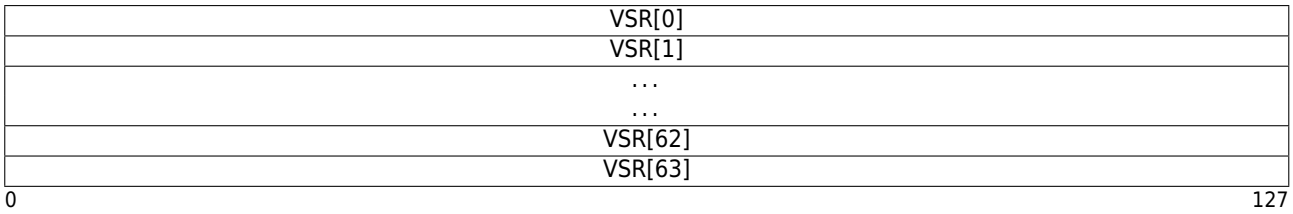


Figure 7.1: Vector-Scalar Registers

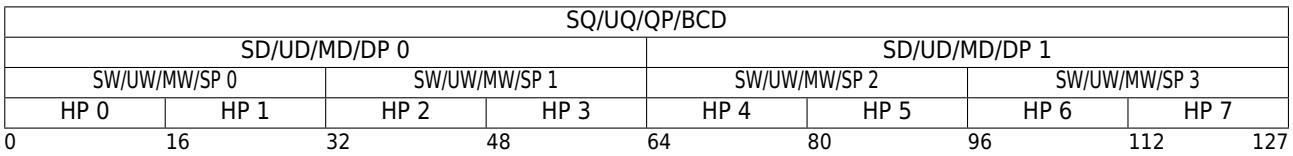


Figure 7.2: Vector-Scalar Register Elements

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 set to 0.

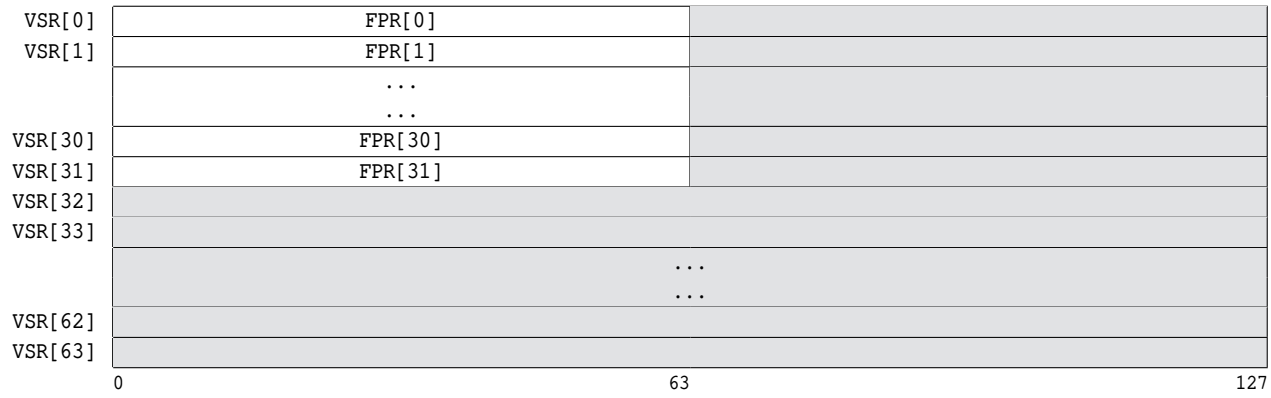


Figure 7.3: Floating-Point Registers as part of VSRs

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.

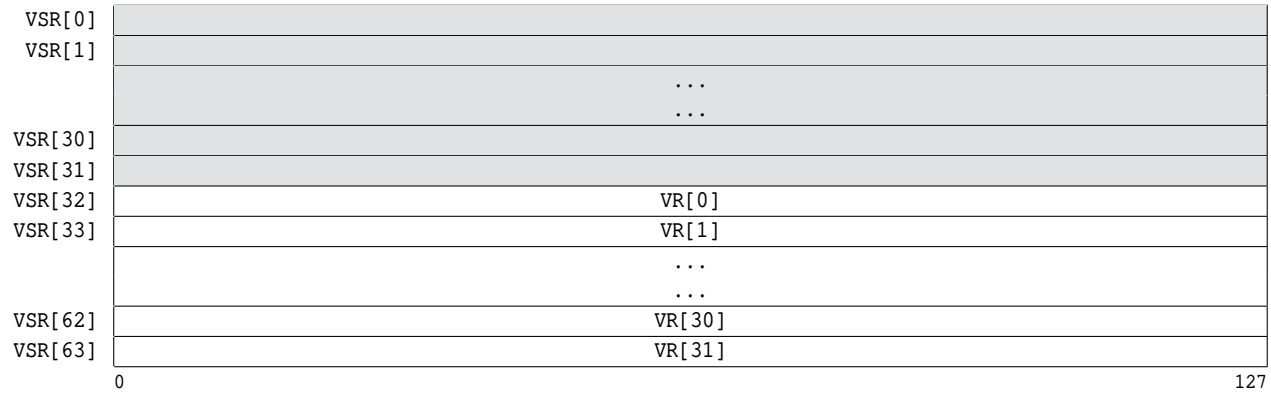


Figure 7.4: Vector Registers as part of VSRs

7.2.1.3 VSX Accumulators

Eight 512-bit Accumulators (ACC) are provided. Each ACC contains four 128-bit rows.

```
ACC[i][0] = ACC[i].bit[0:127]
ACC[i][1] = ACC[i].bit[128:255]
ACC[i][2] = ACC[i].bit[256:383]
ACC[i][3] = ACC[i].bit[384:511]
```

Each ACC is associated with four VSRs in the following manner.

```
ACC[0][0] ↔ VSR[0]
ACC[0][1] ↔ VSR[1]
ACC[0][2] ↔ VSR[2]
ACC[0][3] ↔ VSR[3]

ACC[1][0] ↔ VSR[4]
ACC[1][1] ↔ VSR[5]
ACC[1][2] ↔ VSR[6]
ACC[1][3] ↔ VSR[7]

    :         :

ACC[i][0] ↔ VSR[4×i]
ACC[i][1] ↔ VSR[4×i+1]
ACC[i][2] ↔ VSR[4×i+2]
ACC[i][3] ↔ VSR[4×i+3]

    :         :

ACC[7][0] ↔ VSR[28]
ACC[7][1] ↔ VSR[29]
ACC[7][2] ↔ VSR[30]
ACC[7][3] ↔ VSR[31]
```

While the ACCs are treated as separate registers from the VSRs, ACC[i] may use its associated VSRs 4×i to 4×i+3 as scratch space. That is, when ACC[i] contains defined data, the contents of VSRs 4×i to 4×i+3 are undefined until either a *VSX Move From ACC* instruction is used to copy the contents of ACC[i] to VSRs 4×i to 4×i+3 or some other instruction directly writes to one of these VSRs.

Any instruction that targets any VSR(s) associated with an ACC causes the contents of that ACC to be undefined. If the instruction is not ***xxmfacc***, the target VSR(s) will contain defined data generated by the instruction, and the contents of the other VSRs associated with the ACC will be undefined. If the instruction is ***xxmfacc***, all four VSRs associated with the ACC will contain defined data if the ACC contained defined data, and will have undefined contents otherwise.

Any instruction that targets an ACC causes any subsequent use of that ACC as a source operand to be defined, but causes the contents of all VSRs associated with the ACC to be undefined.

Programming Note

Application software must strictly adhere to the programming model described in this section to guarantee compatibility with future versions of the architecture.

For this version of the architecture, the hardware implementation provides the effect of ACC[i] and VSRs 4×i to 4×i+3 logically containing the same data. Being subject to change in future versions of this architecture, application software must not rely on this behavior. However, system software that handles context save/restore operations need only save and restore data from and to the VSRs (that is, the most current data between the VSRs and associated ACCs will be provided by the hardware implementation). The Accumulators introduce no new logical state at this time. However, future versions of the architecture may define new architectural state or re-define the backing state of the ACC registers. In turn, this may require changes to the system software to support programs written according to this version of the architecture.

The following instructions can be used to copy the contents of an ACC into its associated VSRs.

*VSX Move From Accumulator (**xxmfacc**)*
The contents of the source ACC are copied into the VSRs associated with the target ACC.

The following instructions can be used to initialize the contents of an ACC.

*VSX Move To Accumulator (**xxmtacc**)*
The contents of the VSRs associated with the target ACC are copied into the target ACC.

*VSX Set ACC to Zero (**xxsetaccz**)*
The target ACC is set to 0.

*[Prefixed Masked] VSX Vector 4-bit Signed Integer GER (rank-8) ([**pm**]**xvi4ger8**)*
The sum of the eight outer products of the 4-bit signed integer values in the two vector source operands are placed into the target ACC.

*[Prefixed Masked] VSX Vector 8-bit Signed/Unsigned Integer GER (rank-4) ([**pm**]**xvi8ger4**)*
The sum of the four outer products of the 8-bit signed and unsigned integer values in the two vector source operands are placed into the target ACC.

*[Prefixed Masked] VSX Vector 16-bit Signed Integer GER (rank-2) ([**pm**]**xvi16ger2s**)*
The sum of the two outer products of the 16-bit signed integer values in the two vector source operands are placed into the target ACC.

*[Prefixed Masked] VSX Vector 16-bit Floating-Point GER (rank-2) ([**pm**]**xvf16ger2**)*
The sum of the two outer products of the 16-bit floating-point values in the two vector source operands are placed into the target ACC.

*[Prefixed Masked] VSX Vector bfloat16 GER (rank-2) ([**pm**]**xvbf16ger2**)*
The sum of the two outer products of the bfloat16 values in the two vector source operands are placed into the target ACC.

*[Prefixed Masked] VSX Vector 32-bit Floating-Point GER (rank-1) ([**pm**]**xvf32ger**)*
The outer product of the 32-bit floating-point values in the two vector source operands is placed into the target ACC.

*[Prefixed Masked] VSX Vector 64-bit Floating-Point GER (rank-1) ([**pm**]**xvf64ger**)*
The outer product of the 64-bit floating-point values in the two vector source operands is placed into the target ACC.

7.2.2 Floating-Point Status and Control Register

The Floating-Point Status and Control Register (FPSCR) controls the handling of floating-point exceptions and records status resulting from the floating-point operations. Bits 0:19 and 32:55 are status bits. Bits 56:63 are control bits.

The exception status bits in the FPSCR (bits 35:44, 53:55) are sticky; that is, once set to 1 they remain set to 1 until they are set to 0 by an ***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.

Programming Note

Access to *Move To FPSCR* and *Move From FPSCR* instructions requires FP=1.

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.

Bit(s) Definition

0:28 Reserved

29:31 **Decimal Floating-Point Rounding Control** (DRN)
This field is not used by VSX instructions.

32 **Floating-Point Exception Summary** (FX)
Every floating-point instruction, except ***mtfsfi*** and ***mtfsf***, implicitly sets FX to 1 if that instruction causes any of the floating-point exception bits in the FPSCR to change from 0 to 1. ***mcrfs***, ***mtfsfi***, ***mtfsf***, ***mtfsb0***, and ***mtfsb1*** can alter FX explicitly.

Programming Note

FX is defined not to be altered implicitly by ***mtfsfi*** and ***mtfsf*** because permitting these instructions to alter FX implicitly can cause a paradox. An example is an ***mtfsfi*** or ***mtfsf*** instruction that supplies 0 for FX and 1 for OX, and is executed when OX=0. See also the Programming Notes with the definition of these two instructions.

33 **Floating-Point Enabled Exception Summary** (FEX)
This bit is the OR of all the floating-point exception bits masked by their respective enable bits. ***mcrfs***, ***mtfsfi***, ***mtfsf***, ***mtfsb0***, and ***mtfsb1*** cannot alter FEX explicitly.

34 **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 VX explicitly.

35 **Floating-Point Overflow Exception** (OX)
This bit is set to 1 when a *VSX Scalar Floating-Point Arithmetic*, *VSX Vector Floating-Point Arithmetic*, *VSX Scalar DP-SP Conversion* or *VSX Vector DP-SP Conversion* class instruction causes an Overflow exception. See Section 7.4.3, "Floating-Point Overflow Exception" on page 518.
This bit can be set to 0 or 1 by a *Move To FPSCR* class instruction.

36 **Floating-Point Underflow Exception** (UX)
This bit is set to 1 when a *VSX Scalar Floating-Point Arithmetic*, *VSX Vector Floating-Point Arithmetic*, *VSX Scalar DP-SP Conversion* or *VSX Vector DP-SP Conversion* class instruction causes an Underflow exception. See Section 7.4.4, "Floating-Point Underflow Exception" on page 521.
This bit can be set to 0 or 1 by a *Move To FPSCR* class instruction.

37 **Floating-Point Zero Divide Exception** (ZX)
This bit is set to 1 when a *VSX Scalar Floating-Point Arithmetic* or *VSX Vector Floating-Point Arithmetic* class instruction causes an Zero Divide exception. See Section 7.4.2, "Floating-Point Zero Divide Exception" on page 516.
This bit can be set to 0 or 1 by a *Move To FPSCR* class instruction.

38 **Floating-Point Inexact Exception** (XX)
This bit is set to 1 when a *VSX Scalar Floating-Point Arithmetic*, *VSX Vector Floating-Point Arithmetic*, *VSX Scalar Integer Conversion*, *VSX Vector Integer Conversion*, *VSX Scalar Round to Floating-Point Integer*, or *VSX Vector Round to Floating-Point Integer* class instruction causes an Inexact exception. See Section 7.4.5, "Floating-Point Inexact Exception" on page 525.

This bit can be set to 0 or 1 by a *Move To FPSCR* class instruction.

39 **Floating-Point Invalid Operation Exception (SNAN)** (VXSNAN)

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 508.

This bit can be set to 0 or 1 by a *Move To FPSCR* class instruction.

40 **Floating-Point Invalid Operation Exception (Inf-Inf)** (VXISI)

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 508.

This bit can be set to 0 or 1 by a *Move To FPSCR* class instruction.

41 **Floating-Point Invalid Operation Exception (Inf÷Inf)** (VXIDI)

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 508.

This bit can be set to 0 or 1 by a *Move To FPSCR* class instruction.

42 **Floating-Point Invalid Operation Exception (Zero÷Zero)** (VXZDZ)

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 508.

This bit can be set to 0 or 1 by a *Move To FPSCR* class instruction.

43 **Floating-Point Invalid Operation Exception (Inf×Zero)** (VXIMZ)

This bit is set to 1 when a *VSX Scalar Floating-Point Arithmetic* and *VSX Vector Floating-Point Arithmetic* class instruction causes a Infinity × Zero type Invalid Operation exception. See Section 7.4.1, "Floating-Point Invalid Operation Exception" on page 508.

This bit can be set to 0 or 1 by a *Move To FPSCR* class instruction.

44 **Floating-Point Invalid Operation Exception (Invalid Compare)** (VXVC)

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 508.

This bit can be set to 0 or 1 by a *Move To FPSCR* class instruction.

45 **Floating-Point Fraction Rounded** (FR)

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 499. This bit is not sticky.

46 **Floating-Point Fraction Inexact** (FI)

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 499. This bit is not sticky.

See the definition of XX, above, regarding the relationship between FI and XX.

47:51 **Floating-Point Result Flags** (FPRF)

VSX Scalar Floating-Point Arithmetic, *VSX Scalar DP-SP Conversion*, *VSX Scalar Convert Integer to Double-Precision*, and *VSX Scalar Round to Double-Precision Integer* class instructions set this field based on the result placed into the target register and on the target precision, except that if any portion of the result is undefined then the value placed into FPRF is undefined.

For *VSX Scalar Convert Double-Precision to Integer* class instructions, the value placed into FPRF is undefined.

Additional details are as follows.

Result Flags					Result Value Class
C	FL	FG	FE	FU	
1	0	0	0	1	Quiet NaN
0	1	0	0	1	− Infinity
0	1	0	0	0	− Normalized Number
1	1	0	0	0	− Denormalized Number
1	0	0	1	0	− Zero
0	0	0	1	0	+ Zero
1	0	1	0	0	+ Denormalized Number
0	0	1	0	0	+ Normalized Number
0	0	1	0	1	+ Infinity

Table 7.1: Floating-Point Result Flags

47 **Floating-Point Result Class Descriptor** (C)

VSX Scalar Floating-Point Arithmetic, *VSX Scalar DP-SP Conversion*, *VSX Scalar Convert Integer to Double-Precision*, and *VSX Scalar Round to Double-Precision Integer* class instructions set this bit with the FPCC bits, to indicate the class of the result as shown in Table 7.1, "Floating-Point Result Flags," on page 489.

48:51 **Floating-Point Condition Code** (FPCC)

VSX Scalar Compare Double-Precision instruction sets one of the FPCC bits to 1 and the other three FPCC bits to 0 based on the relative values of the operands being compared.

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 FPCC bits with the C bit, to indicate the class of the result as shown in Table 7.1, "Floating-Point Result Flags," on page 489. Note that in this case the high-order three bits of the FPCC retain their relational significance indicating that the value is less than, greater than, or equal to zero.

48 **Floating-Point Less Than or Negative** (FL)

49 **Floating-Point Greater Than or Positive** (FG)

50 **Floating-Point Equal or Zero** (FE)

51 **Floating-Point Unordered or NaN** (FU)

52 Reserved

53 **Floating-Point Invalid Operation Exception (Software-Defined Condition)** (VXSOFT)

This bit can be altered only by ***mcrfs***, ***mtfsfi***, ***mtfsf***, ***mtfsb0***, or ***mtfsb1***. See Section 7.4.1, "Floating-Point Invalid Operation Exception" on page 508.

Programming Note

VXSOFT can be used by software to indicate the occurrence of an arbitrary, software-defined, condition that is to be treated as an Invalid Operation exception. For example, the bit could be set by a program that computes a base 10 logarithm if the supplied input is negative.

54 **Floating-Point Invalid Operation Exception (Invalid Square Root)** (VXSQRT)

This bit is set to 1 when a *VSX Scalar Floating-Point Arithmetic* or *VSX Vector Floating-Point Arithmetic* class instruction causes a Invalid Square Root type Invalid Operation exception. See Section 7.4.1, "Floating-Point Invalid Operation Exception" on page 508.

This bit can be set to 0 or 1 by a *Move To FPSCR* class instruction.

55 **Floating-Point Invalid Operation Exception (Invalid Integer Convert)** (VXCVI)

This bit is set to 1 when a *VSX Scalar Convert Double-Precision to Integer*, *VSX Vector Convert Double-Precision to Integer*, or *VSX Vector Convert Single-Precision to Integer* class instruction causes a Invalid Integer Convert type Invalid Operation exception. See Section 7.4.1, "Floating-Point Invalid Operation Exception" on page 508.

This bit can be set to 0 or 1 by a *Move To FPSCR* class instruction.

56 **Floating-Point Invalid Operation Exception Enable** (VE)

This bit is used by *VSX Scalar Floating-Point* and *VSX Vector Floating-Point* class instructions to enable trapping on Invalid Operation exceptions. See Section 7.4.1, "Floating-Point Invalid Operation Exception" on page 508.

57 **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 518.

58 **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 521.

59 **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 516.

60 **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 525.

61 **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 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.

62:63 **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 499.

00 Round to Nearest Even
01 Round toward Zero
10 Round toward +Infinity
11 Round toward -Infinity

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.

[]

Computational instructions are defined to be 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.

[]

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 $2^{exponent}$. 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 (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 7.2.2, "Floating-Point Status and Control Register" on page 488 for a description of these exception and enable bits, and Section 7.3.3, "VSX Floating-Point Execution Models" on page 502 for a detailed discussion of floating-point exceptions, including the effects of the enable bits.

7.3.2 VSX Floating-Point Data

7.3.2.1 Data Format

This architecture defines the representation of a floating-point value in five different binary fixed-length formats, 16-bit half-precision format, 16-bit bfloat16 format, 32-bit single-precision format, 64-bit double-precision format, and 128-bit quad-precision format. The half-precision format is used for half-precision floating-point data in storage and registers. The bfloat16 format is used for bfloat16 floating-point data in storage and registers. The single-precision format is used for single-precision floating-point data in storage and registers. The double-precision format is used for double-precision floating-point data in storage and registers. The quad-precision format is used for quad-precision floating-point data in storage and registers.

The lengths of the exponent and the fraction fields differ between these five formats. The structure of the half-precision, bfloat16, single-precision, double-precision, and quad-precision formats is shown in Figure 7.5, Figure 7.7, Figure 7.8, Figure 7.9, and Figure 7.9, respectively.

Values in floating-point format are composed of three fields:

S	sign bit
EXP	exponent+bias
FRACTION	fraction

Representation of numeric values in the floating-point formats consists of a sign bit (S), a biased exponent (EXP), and the fraction portion (FRACTION) of the significand. The significand consists of a leading implied bit concatenated on the right with the FRACTION. This leading implied bit is 1 for normalized numbers and 0 for denormalized (subnormal) numbers or zero and is located in the unit bit position (that is, the first bit to the left of the binary point). Values representable within the three floating-point formats can be specified by the parameters listed in Table 7.2.

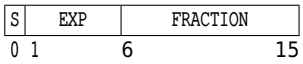


Figure 7.5: Binary floating-point half-precision format (binary16)

Figure 7.6: Binary floating-point bfloat16 format (bfloat16)

Figure 7.7: Binary floating-point single-precision format (binary32)

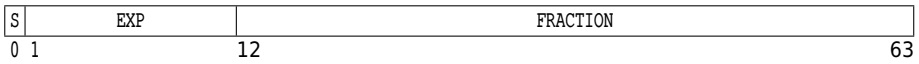


Figure 7.8: Binary floating-point double-precision format (binary64)

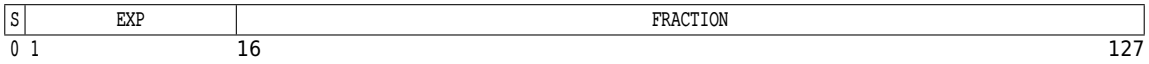


Figure 7.9: Binary floating-point quad-precision format (binary128)

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 7.10.

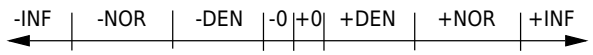


Figure 7.10: Approximation to real numbers

The NaNs are not related to the numeric values or infinities by order or value but are encodings used to convey diagnostic information such as the representation of uninitialized variables.

The following is a description of the different floating-point values defined in the architecture:

Binary floating-point numbers

Machine representable values used as approximations to real numbers. Three categories of numbers are supported: normalized numbers, denormalized numbers, and zero values.

Normalized numbers (±NOR)

These are values that have a biased exponent value in the range:

1 to 30 in half-precision format
1 to 254 in bfloat16 format
1 to 254 in single-precision format
1 to 2046 in double-precision format
1 to 32766 in quad-precision format

	bfloat16	binary16	binary32	binary64	binary128
Exponent Bias	+127	+15	+127	+1023	+16383
Maximum Exponent (Emax)	+127	+15	+127	+1023	+16383
Minimum Exponent (Emin)	−126	−14	−126	−1022	−16382
Widths (bits):					
Format	16	16	32	64	128
Sign	1	1	1	1	1
Exponent	8	5	8	11	15
Fraction	7	10	23	52	112
Significand	8	11	24	53	113
Nmax	$(1-2^{-8})\times 2^{128}$ $\approx 3.4\times 10^{38}$	$(1-2^{-11})\times 2^{16}$ $\approx 6.6\times 10^{4}$	$(1-2^{-24})\times 2^{128}$ $\approx 3.4\times 10^{38}$	$(1-2^{-53})\times 2^{1024}$ $\approx 1.8\times 10^{308}$	$(1-2^{-113})\times 2^{16384}$ $\approx 1.2\times 10^{4932}$
Nmin	1.0×2^{-126} $\approx 1.2\times 10^{-38}$	1.0×2^{-14} $\approx 6.1\times 10^{-5}$	1.0×2^{-126} $\approx 1.2\times 10^{-38}$	1.0×2^{-1022} $\approx 2.2\times 10^{-308}$	1.0×2^{-16382} $\approx 3.4\times 10^{-4932}$
Dmin	1.0×2^{-133} $\approx 9.2\times 10^{-41}$	1.0×2^{-24} $\approx 6.0\times 10^{-8}$	1.0×2^{-149} $\approx 1.4\times 10^{-45}$	1.0×2^{-1074} $\approx 4.9\times 10^{-324}$	1.0×2^{-16494} $\approx 6.5\times 10^{-4966}$

≈	Value is approximate
Dmin	Smallest (in magnitude) representable denormalized number.
Nmax	Largest (in magnitude) representable number.
Nmin	Smallest (in magnitude) representable normalized number.

Table 7.2: Binary floating-point fields

They are values in which the implied unit bit is 1. Normalized numbers are interpreted as follows:

```
NOR = (-1)^s × 2^E × (1.fraction)
```

where `s` is the sign, `E` is the unbiased exponent, and `1.fraction` is the significand, which is composed of a leading unit bit (implied bit) and a fraction part.

Zero values (±0)

These are values that have a biased exponent value of zero and a fraction value of zero. Zeros can have a positive or negative sign. The sign of zero is ignored by comparison operations (that is, comparison regards +0 as equal to −0).

Denormalized numbers (±DEN)

These are values that have a biased exponent value of zero and a nonzero fraction value. They are nonzero numbers smaller in magnitude than the representable normalized numbers. They are values in which the implied unit bit is 0. Denormalized numbers are interpreted as follows:

```
DEN = (-1)^s × 2^Emin × (0.fraction)
```

where `Emin` is the minimum representable exponent value.

-14	for half-precision
-126	for bfloat16
-126	for single-precision
-1022	for double-precision
-16382	for quad-precision.

Infinities (±INF)

These are values that have the maximum biased exponent value:

31	in half-precision format
255	in bfloat16 format
255	in single-precision format
2047	in double-precision format
32767	in quad-precision format

and a zero fraction value. They are used to approximate values greater in magnitude than the maximum normalized value.

Infinity arithmetic is defined as the limiting case of real arithmetic, with restricted operations defined among numbers and infinities. Infinities and the real numbers can be related by ordering in the affine sense:

−Infinity < every finite number < +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 508.

For comparison operations, +Infinity compares equal to +Infinity and −Infinity compares equal to −Infinity.

Not a Numbers (NaNs)

These are values that have the maximum biased exponent value and a nonzero fraction value. The sign bit is ignored (that is, NaNs are neither positive nor negative). If the high-order bit of the fraction field is 0, the NaN is a *Signaling NaN*; otherwise it is a *Quiet NaN*.

Signaling NaNs are used to signal exceptions when they appear as operands of computational instruc-

tions.

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(src1,src3,src2)
   ex: result = (src1 x src3) - src2

f(src1,src2)
   ex: result = src1 x src2
   ex: result = src1 + src2

f(src1)
   ex: result = f(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.

```
if src1 is a NaN
  then result = Quiet(src1)
  else if src2 is a NaN (if there is a src2)
    then result = Quiet(src2)
    else if src3 is a NaN (if there is a src3)
      then result = Quiet(src3)
      else if disabled invalid operation exception
        then result = generated QNaN
```

where Quiet(x) means x if x is a QNaN and x converted to a QNaN if x is an SNaN. Any instruction that generates a QNaN as the result of a disabled Invalid Operation exception generates the value,

0x7E00 for half-precision results,
0x7FC0 for bfloat16 results,
0x7FC0_0000 for single-precision results,
0x7FF8_0000_0000_0000 for double-precision results,
0x7FFF_8000_0000_0000_0000_0000_0000_0000 for quad-precision results.

Note that the M-form multiply-add-type instructions use the B source operand to specify src3 and the T target operand to specify src2, whereas A-form multiply-add-type instructions use the B source operand to specify src2 and the T target operand to specify src3.

A double-precision NaN is considered to be representable in single-precision format if and only if the low-order 29 bits of the double-precision NaN's fraction are zero.

7.3.2.3 Sign of Result

The following rules govern the sign of the result of an arithmetic, rounding, or conversion operation, when the operation does not yield an exception. They apply even when the operands or results are zeros or infinities.

- The sign of the result of an add operation is the sign of the operand having the larger absolute value. If both operands have the same signs, the sign of the result of an add operation is the same as the sign of the operands. The sign of the result of the subtract operation $x - y$ is the same as the sign of the result of the add operation $x + (-y)$.

  When the sum of two operands with opposite sign, or the difference of two operands with the same signs, is exactly zero, the sign of the result is positive in all rounding modes except Round toward $-$Infinity, in which mode the sign is negative.
- The sign of the result of a multiply or divide operation is the Exclusive OR of the signs of the operands.
- The sign of the result of a *Square Root* or *Reciprocal Square Root Estimate* operation is always positive, except that the square root of -0 is -0 and the reciprocal square root of -0 is $-$Infinity.
- The sign of the result of a *Convert From Integer* or *Round to Floating-Point Integer* operation is the sign of the operand being converted.

For the *Multiply-Add* instructions, the rules given above are applied first to the multiply operation and then to the add or subtract operation (one of the inputs to the add or subtract operation is the result of the multiply operation).

7.3.2.4 Normalization and Denormalization

The intermediate result of an arithmetic instruction can require normalization and/or denormalization as described below. Normalization and denormalization do not affect the sign of the result.

When an arithmetic or rounding instruction produces an intermediate result which carries out of the significand, or in which the significand is nonzero but has a leading zero bit, it is not a normalized number and must be normalized before it is stored. For the carry-out case, the significand is shifted right one bit, with a one shifted into the leading significand bit, and the exponent is incremented by one. For the leading-zero case, the significand is shifted left while decrementing its exponent by one for each bit shifted, until the leading significand bit becomes one. The Guard bit and the Round bit (see Section 7.3.3.1, "VSX Execution Model for IEEE Operations" on page 502) 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 521. 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 521) and Underflow exception is signaled.

Engineering Note

When denormalized numbers are operands of multiply, divide, and square root operations, some implementations might prenormalize the operands internally before performing the operations.

7.3.2.5 Data Handling and Precision

Scalar double-precision floating-point data is represented in double-precision format in VSRs and storage.

Vector double-precision floating-point data is represented in double-precision format in VSRs and storage.

Scalar single-precision floating-point data is represented in double-precision format in VSRs and in single-precision format in storage.

Vector single-precision floating-point data is represented in single-precision format in VSRs and storage.

Double-precision operands may be used as input for double-precision scalar arithmetic operations.

Double-precision operands may be used as input for single-precision scalar arithmetic operations when trapping on overflow and underflow exceptions is disabled.

Single-precision operands may be used as input for double-precision and single-precision scalar arithmetic operations.

Double-precision operands may be used as input for double-precision vector arithmetic operations.

Single-precision operands may be used as input for single-precison 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.

bfloat16 Operands

Instructions are provided to convert between bfloat16 and single-precision formats for vector data in VSRs.

An instruction is provided to explicitly convert bfloat16 format operands in a VSR to single-precision format.

1. *VSX Vector Convert Half-Precision to Single-Precision format XX2-form*

   The bfloat16 floating-point value in the rightmost halfword of each word element of the source VSR is placed into the corresponding word element of the target VSR in single-precision format.

2. *VSX Vector Convert with round Single-Precision to Half-Precision format XX2-form*
   The single-precision floating-point value in each word element i of the source VSR is rounded to bfloat16 precision and placed into the rightmost halfword of the corresponding word element of the target VSR in bfloat16 format.

Single-Precision Operands

For single-precision scalar data, a conversion from single-precision format to double-precision format is performed when loading from storage into a VSR and a conversion from double-precision format to single-precision format is performed when storing from a VSR to storage. No floating-point exceptions are caused by these instructions.

Instructions are provided to convert between single-precision and double-precision formats for scalar and vector data in VSRs.

An instruction is provided to explicitly convert a double format operand in a VSR to single-precision. Scalar single-precision floating-point is enabled with six types of instructions.

1. *Load VSX Scalar Single-Precision*
   [***p***]***lxssp*** and ***lxsspx*** access a floating-point operand in single-precision format in storage, convert it to double-precision format, and load it into a VSR. No floating-point exceptions are caused by these instructions.
2. *VSX Scalar Round to Single-Precision XX2-form*
   xsrsp rounds a double-precision operand to single-precision, checking the exponent for single-precision range and handling any exceptions according to respective enable bits, and places that operand into a VSR in double-precision format. For results produced by single-precision arithmetic instructions, single-precision loads, and other instances of ***xsrsp***, ***xsrsp*** 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. *VSX Scalar Convert Single-Precision to Double-Precision format*
   VSX Scalar Convert Single-Precision to Double-Precision format XX2-form (***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.
   VSX Scalar Convert Single-Precision to Double-Precision format Non-signalling XX2-form (***xscvspdpn***) accesses a floating-point operand in single-precision format from word element 0 of the source VSR, converts it to double-precision format, and places it into doubleword element 0 of the target VSR. ***xscvspdpn*** does not set any exception status (i.e., VXSNAN).
4. *VSX Scalar Convert Double-Precision to Single-Precision format* [*Non-Signalling*]
   VSX Scalar Convert with round Double-Precision to Single-Precision format XX2-form (***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 elements 0 and 1 of the target VSR in single-precision format. This function would be used to port scalar floating-point data to a format compatible for single-precision vector operations. Values greater in magnitude than 2^{319} when Overflow is enabled (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.
   VSX Scalar Convert Scalar Single-Precision to Vector Single-Precision format Non-signalling XX2-form (***xscvdpspn***) directly converts the single-precision floating-point value represented in double-precision format in doubleword element 0 of the source VSR to single-precision format, without rounding, and places the result into word elements 0 and 1 of the target VSR in single-precision format. ***xscvdpspn*** does not set any exception status (i.e., VXSNAN).
5. *VSX Scalar Single-Precision Arithmetic*[1]
   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*

[1] *VSX Scalar Single-Precision Arithmetic* instructions: ***xsaddsp***, ***xsdivsp***, ***xsmulsp***, ***xsresp***, ***xssubsp***, ***xsmaddasp***, ***xsmaddmsp***, ***xsmsubasp***, ***xsmsubmsp***, ***xsnmaddasp***, ***xsnmaddmsp***, ***xsnmsubasp***, ***xsnmsubmsp***

Store VSX Scalar Single-Precision DS-form (***stxssp***), Prefixed Store VSX Scalar Single-Precision 8LS:D-form (***pstxssp***), and *Store VSX Scalar Single-Precision Indexed X-form* (***stxsspx***) convert a single-precision value that is in double-precision format to single-precision format and stores that operand into storage. No floating-point exceptions are caused by these instructions. (The value being stored is effectively assumed to be the result of an instruction of one of the preceding five types.)

When the result of a *Load VSX Scalar Single-Precision* (***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 respresentable 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 (***xscmpexpdp***, ***xsiexpdp***, ***xststdcdp***, ***xststdcsp***, ***xsxexpdp***, ***xsxsigdp***) can be used to perform these operations on scalar single-precision operands.

Integer-Valued Operands

Instructions are provided to round floating-point operands to integer values in floating-point format. To facilitate exchange of data between the floating-point and integer processing, instructions are provided to convert between floating-point double and single-precision format and integer word and doubleword format in a VSR. Computation on integer-valued operands can be performed using arithmetic instructions of the required precision. (The results might not be integer values.) The three groups of instructions provided specifically to support integer-valued operands are described below.

1. Rounding to a floating-point integer

   VSX Scalar Round to Double-Precision Integer[2] 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[3] instructions round each double-precision vector operand element to an integer value in double-precision format.

   VSX Vector Round to Single-Precision Integer[4] instructions round each single-precision vector

[2] *VSX Scalar Round to Double-Precision Integer* instructions: ***xsrdpi***, ***xsrdpip***, ***xsrdpim***, ***xsrdpiz***, ***xsrdpic***
[3] *VSX Vector Round to Double-Precision Integer* instructions: ***xvrdpi***, ***xvrdpip***, ***xvrdpim***, ***xvrdpiz***, ***xvrdpic***
[4] *VSX Vector Round to Single-Precision Integer* instructions: ***xvrspi***, ***xvrspip***, ***xvrspim***, ***xvrspiz***, ***xvrspic***

operand element to an integer value in single-precision format.

Except for ***xsrdpic***, ***xvrdpic***, and ***xvrspic***, rounding is performed using the rounding mode specified by the opcode. For ***xsrdpic***, ***xvrdpic***, and ***xvrspic***, rounding is performed using the rounding mode specified by RN.

VSX Round to Floating-Point Integer[5] instructions can cause Invalid Operation (VXSNAN) exceptions.

xsrdpic, ***xvrdpic***, and ***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[6] 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[7] 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[8] instructions converts the single-precision value in each odd-numbered word element of the source vector operand to a 64-bit signed or unsigned integer format.

   VSX Vector Single-Precision to Integer Word Format Conversion[9] instructions converts the single-precision value in each word element of the source vector operand to either a 32-bit signed or unsigned integer format.

   Rounding is performed using Round Towards Zero rounding mode. These instructions can cause Invalid Operation (VXSNAN, VXCVI) and Inexact exceptions.

3. Converting integer format to floating-point format

   VSX Scalar Integer Doubleword to Double-Precision Format Conversion[10] 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[11] 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[12] instructions converts the 64-bit signed or unsigned integer in each doubleword element in the source vector operand to double-precision floating-point format.

   VSX Vector Integer Doubleword to Single-Precision Format Conversion[13] instructions convert the 64-bit signed or unsigned integer in each doubleword element in the source vector operand to single-precision floating-point format.

   VSX Vector Integer Word to Single-Precision Format Conversion[14] instructions convert the 32-bit signed or unsigned integer in each word element in the source vector operand to single-precision floating-point format.

   Rounding is performed using the rounding mode specificed 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 half-precision, bfloat16, single-precision, double-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

[5] *VSX Round to Floating-Point Integer* instructions: ***xsrdpi***, ***xsrdpip***, ***xsrdpim***, ***xsrdpiz***, ***xsrdpic***, ***xvrdpi***, ***xvrdpip***, ***xvrdpim***, ***xvrdpiz***, ***xvrdpic***, ***xvrspi***, ***xvrspip***, ***xvrspim***, ***xvrspiz***, and ***xvrspic***

[6] *VSX Scalar Double-Precision to Integer Format Conversion* instructions: ***xscvdpsxds***, ***xscvdpsxws***, ***xscvdpuxds***, ***xscvdpuxws***

[7] *VSX Vector Double-Precision to Integer Format Conversion* instructions: ***xvcvdpsxds***, ***xvcvdpsxws***, ***xvcvdpuxds***, ***xvcvdpuxws***

[8] *VSX Vector Single-Precision to Integer Doubleword Format Conversion* instructions: ***xvcvspsxds***, ***xvcvspuxds***

[9] *VSX Vector Single-Precision to Integer Word Format Conversion* instructions: ***xvcvspsxws***, ***xvcvspuxws***

[10] *VSX Scalar Integer Doubleword to Double-Precision Format Conversion* instructions: ***xscvsxddp***, ***xscvuxddp***

[11] *VSX Scalar Integer Doubleword to Single-Precision Format Conversion* instructions: ***xscvsxdsp***, ***xscvuxdsp***

[12] *VSX Vector Integer Doubleword to Double-Precision Format Conversion* instructions: ***xscvsxddp***, ***xscvuxddp***

[13] *VSX Vector Integer Doubleword to Single-Precision Format Conversion* instructions: ***xscvsxdsp***, ***xscvuxdsp***

[14] *VSX Vector Integer Word to Single-Precision Format Conversion* instructions: ***xscvsxwsp***, ***xscvuxwsp***

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 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 incrementing 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.

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 488. These are encoded as follows.

RN	Rounding Mode
00	Round to Nearest Even
01	Round towards Zero
10	Round towards +Infinity
11	Round towards -Infinity

A fifth rounding mode is provided in the round to floating-point integer instructions (Section 7.6.2.8.2 on page 808), 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,

```
xsaddqpo   Tx,A,B        ; use Round to Odd override (RO=1)
xsrqpxp    Tdxp,Tx       ; final QP result rounded to DXP
```

To return a quad-precision result rounded to double-precision requires a 3-instruction sequence,

```
xsaddqpo   Tx,A,B        ; use Round to Odd override (RO=1)
xscvqpdp   Temp,Tx       ; QP 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,

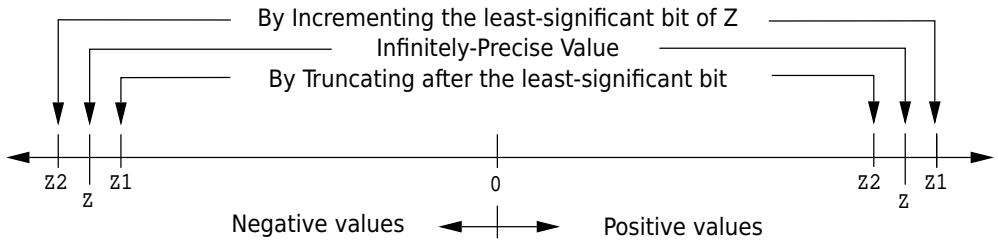
```
xsaddqpo   Tx,A,B        ; use Round to Odd override (RO=1)
xscvqpdpo  Temp,Tx       ; QP result rounded to DP using Round to Odd & converted to DP format
xsrsp      Temp,Temp     ; DP result is rounded to SP
xscvdpqp   Tsp,Temp      ; final QP result rounded to SP
```

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 7.11 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 502 for a detailed explanation of rounding.

Figure 7.11 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 (z1 or z2). In case of a tie, choose the one that is furthest away from 0.

Round to Nearest Even

Choose z if z is representable in the target precision.

Otherwise, choose the value that is closer to z (z1 or z2). In case of a tie, choose the one that is even (least significant bit is 0).

Round to Odd

Choose z if z is representable in the target precision.

Otherwise, choose the value (z1 or z2) that is odd (least significant bit is 1).

Round toward Zero

Choose z if z is representable in the target precision.

Otherwise, choose the smaller in magnitude (z1 or z2).

Round toward +Infinity

Choose z if z is representable in the target precision.

Otherwise, choose z1.

Round toward −Infinity

Choose z if z is representable in the target precision.

Otherwise, choose z2.

Figure 7.11: Selection of Z1 and Z2

7.3.3 VSX Floating-Point Execution Models

All implementations of this architecture must provide the equivalent of the following execution models to ensure that identical results are obtained.

Special rules are provided in the definition of the computational instructions for the infinities, denormalized numbers and NaNs. The material in the remainder of this section applies to instructions that have numeric operands and a numeric result (that is, operands and result that are not infinities or NaNs), and that cause no exceptions. See Section 7.3.2.2 and Section 7.3.3 for the cases not covered here.

Although the double-precision format specifies an 11-bit exponent, exponent arithmetic makes use of two additional bits to avoid potential transient overflow and underflow conditions. One extra bit is required when denormalized double-precision numbers are prenormalized. The second bit is required to permit the computation of the adjusted exponent value in the following cases when the corresponding exception enable bit is 1:

- Underflow during multiplication using a denormalized operand.
- Overflow during division using a denormalized divisor.
- Undeflow during division using denormalized dividend and a large divisor.

The IEEE standard includes 32-bit and 64-bit arithmetic. The standard requires that single-precision arithmetic be provided for single-precision operands.

VSX defines both scalar and vector double-precision floating-point operations to operate only on double-precision operands. VSX also defines vector single-precision floating-point operations to operate only on single-precision operands.

7.3.3.1 VSX Execution Model for IEEE Operations

IEEE-conforming significand arithmetic is considered to be performed with a floating-point accumulator having the following format, where bits `0:p-1` comprise the significand of the intermediate result (where `p` is the length of the significand).

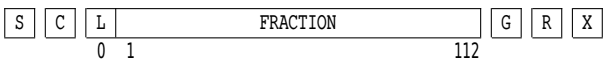


Figure 7.12: IEEE quad-precision (binary128) floating-point execution model (p=113)

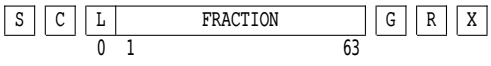


Figure 7.13: IEEE double-extended-precision floating-point execution model (p=64)

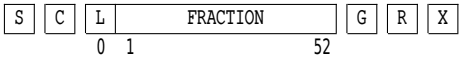


Figure 7.14: IEEE double-precision (binary64) floating-point execution model (p=53)

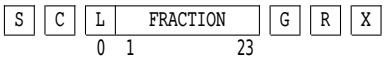


Figure 7.15: IEEE single-precision (binary32) floating-point execution model (p=24)

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 7.3 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).

G	R	X	Interpretation
0	0	0	IR is exact
0	0	1	IR closer to NL
0	1	0	
0	1	1	
1	0	0	IR midway between NL and NH
1	0	1	IR closer to NH
1	1	0	
1	1	1	

Table 7.3: Interpretation of G, R, and X bits

Table 7.4 shows the positions of the Guard, Round, and Sticky bits for quad-precision, double-extended precision, double-precision and single-precision floating-point numbers relative to the accumulator illustrated in Figures 7.12, 7.13, 7.14, and 7.15.

Format	Guard	Round	Sticky
Double	G bit	R bit	X bit
Single	24	25	OR of bits 26:52, G, R, X

Table 7.4: Location of the Guard, Round, and Sticky bits in the IEEE execution model

The significand of the intermediate result is prepared for rounding by shifting its contents right, if required, until the least significant bit to be retained is in the low-order bit position of the fraction.

Six rounding modes are provided as described in Section 7.3.2.6, “Rounding” on page 499. The rules for rounding in each mode are as follows.

- ***Round to Nearest Even***
  If IR is exact, choose IR.
  Otherwise, if IR is closer to NL, choose NL.
  Otherwise, if IR is closer to NH, choose NH.
  Otherwise, if IR is midway between NL and NH, choose whichever of NL and NH is even.
- ***Round towards Zero***
  If IR is exact, choose IR.
  Otherwise, choose NL.
- ***Round towards +Infinity***
  If IR is exact, choose IR.
  Otherwise, if positive, choose NH.
  Otherwise, if negative, choose NL.
- ***Round towards −Infinity***
  If IR is exact, choose IR.
  Otherwise, if positive, choose NL.
  Otherwise, if negative, 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 to Odd***
  If IR is exact, choose IR.
  Otherwise, choose NL, and if G=1, R=1, or X=1, the least-significant bit of the result is set to 1.

Four of the rounding modes are user-selectable through RN.

RN	Rounding Mode
0b00	Round to Nearest Even
0b01	Round toward Zero
0b10	Round toward +Infinity
0b11	Round toward −Infinity

Round to Nearest Away is provided in the *VSX Round to Floating-Point Integer* instructions (Section 7.6.2.8.2 on page 808).

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.

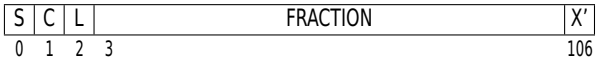


Figure 7.16: Multiply-add 64-bit execution model

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 7.5

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.

Format	Guard	Round	Sticky
Double	53	54	OR of 55:105, X’
Single	24	25	OR of 26:105, X’

Table 7.5: 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. Instructions that modify Accumulators produce results as if all exceptions are disabled. For other VSX instructions, if an exception occurs, the corresponding enable bit governs the result produced by the instruction. For all VSX instructions in which an exception occurs, the corresponding enable bit in conjunction with the FE0 and FE1 bits (see page 506) governs 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 and type of instruction.

Instructions that modify Accumulators always write the target register. For other VSX instructions, the writing of a result to the target register is suppressed for 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 from the VSX instructions that don't modify Accumulators 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 (for instructions other than those which update Accumulators). Table 7.6 lists the types of exceptions and indicates whether a result is written to the target VSR or suppressed.

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, instructions that modify Accumulators always generate the default result value with the expectation that if the exception is enabled, software will emulate the instruction with scalar instructions to get the desired result. For instructions that do not modify Accumulators, 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 that software will revise the result. For instructions that do not modify Accumulators, 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, with the expectation that the exception is not detected by software and the default result will be used. The result to be delivered in each case for each exception is described in the following sections.

On exception type...	Scalar Instruction Results	Vector Instruction Non-Accumulator Results	Vector Instruction Accumulator Results
Enabled Invalid Operation	suppressed	suppressed	written
Enabled Zero Divide	suppressed	suppressed	written
Enabled Overflow	written	suppressed	written
Enabled Underflow	written	suppressed	written
Enabled Inexact	written	suppressed	written
Disabled Invalid Operation	written	written	written
Disabled Zero Divide	written	written	written
Disabled Overflow	written	written	written
Disabled Underflow	written	written	written
Disabled Inexact	written	written	written

Table 7.6: Exception Types Result Suppression

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
0	0	**Ignore Exceptions Mode** Floating-point exceptions do not cause the system floating-point enabled exception error handler to be invoked.
0	1	**Imprecise Nonrecoverable Mode** 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.
1	0	**Imprecise Recoverable Mode** 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.
1	1	**Precise Mode** The system floating-point enabled exception error handler is invoked precisely at the instruction that caused the enabled exception.

Table 7.7: Floating-point exception modes

For all floating-point instructions except those that specify an Accumulator as the target, 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 the subsections of this section. For floating-point instructions that specify an Accumulator as the target, floating-point results are computed as if all FPSCR exception enable bits are set to zero, and are always stored in the target Accumulator, regardless of the contents of the FPSCR exception enable bits. In all cases, the question of whether a floating-point result is stored, and what value is stored, is not affected by the value of the FE0 and FE1 bits.

The subsections of this section do not explicitly cover the floating-point instructions that specify an Accumulator as the target. For those instructions, the subsections are to be interpreted as applying independently to the computation of the result for each element of the target Accumulator assuming the corresponding enable bit is zero, where the meaning of “element” is implied by the name of the instruction.

In all cases in which the system floating-point enabled exception error handler is invoked, all instructions before the instruction at which the system floating-point enabled exception error handler is invoked have been completed, and no instruction after the instruction at which the system floating-point enabled exception error handler is invoked has begun execution. The instruction at which the system floating-point enabled exception error handler is invoked has completed if it is the excepting instruction, and there is only one such instruction. Otherwise, it has not begun execution, or has been partially executed in some cases, as described in Book III.

Programming Note

In any of the three non-Precise modes, a *Floating-Point Status and Control Register* instruction can be used to force any exceptions, because of instructions initiated before the *Floating-Point Status and Control Register* instruction, to be recorded in the FPSCR. (This forcing is superfluous for Precise Mode.)

In both Imprecise modes, a *Floating-Point Status and Control Register* instruction can be used to force any invocations of the system floating-point enabled exception error handler that result from instructions initiated before the *Floating-Point Status and Control Register* instruction to occur. This forcing has no effect in Ignore Exceptions Mode, and is superfluous for Precise Mode.

The last sentence of the paragraph preceding this Programming Note can apply only in the Imprecise modes, or if the mode has just been changed from Ignore Exceptions Mode to some other mode. It always applies in the latter case.

To obtain the best performance across the widest range of implementations, the programmer should obey the following guidelines.

- If the IEEE default results are acceptable to the application, Ignore Exceptions Mode should be used with all FPSCR exception enable bits set to 0.
- If the IEEE default results are not acceptable to the application, Imprecise Nonrecoverable Mode should be used, or Imprecise Recoverable Mode if recoverability is needed, with FPSCR exception enable bits set to 1 for those exceptions for which the system floating-point enabled exception error handler is to be invoked.
- Ignore Exceptions Mode should not, in general, be used when any FPSCR exception enable bits are set to 1.
- Precise Mode can degrade performance in some implementations, perhaps substantially, and therefore should be used only for debugging and other specialized applications.

Engineering Note

It is permissible for the implementation to be precise in any of the three modes that permit interrupts, or to be recoverable in Nonrecoverable Mode.

7.4.1 Floating-Point Invalid Operation Exception

7.4.1.1 Definition

An Invalid Operation exception occurs when an operand is invalid for the specified operation. The invalid operations are:

SNaN
Any floating-point operation on a Signaling NaN.

Infinity–Infinity
Magnitude subtraction of infinities.

Infinity÷Infinity
Floating-point division of infinity by infinity.

Zero÷Zero
Floating-point division of zero by zero.

Infinity×Zero
Floating-point multiplication of infinity by zero.

Invalid Compare
Floating-point ordered comparison involving a NaN.

Invalid Square Root
Floating-point square root or reciprocal square root of a nonzero negative number.

Invalid Integer Convert
Floating-point-to-integer convert involving a number too large in magnitude to be represented in the target format, or involving an infinity or a NaN.

An Invalid Operation exception also occurs when an ***mtfsfi***, ***mtfsf***, or ***mtfsb1*** instruction is executed that sets VXSOFT to 1 (Software-Defined Condition).

The action to be taken depends on the setting of the Invalid Operation Exception Enable bit of the FPSCR.

7.4.1.2 Action for VE=1

When Invalid Operation exception is enabled (VE=1) and an Invalid Operation exception occurs, the following actions are taken:

- For any of the following instructions,
  VSX Scalar Floating-Point Arithmetic instructions
  VSX Scalar DP-SP Conversion instructions
  VSX Scalar Convert Floating-Point to Integer instructions
  VSX Scalar Round to Floating-Point Integer instructions
  do the following.
  1. One or two of the following Invalid Operation exceptions are set to 1.

VXSNAN	(if SNaN)
VXISI	(if Infinity–Infinity)
VXIDI	(if Infinity÷Infinity)
VXZDZ	(if Zero÷Zero)
VXIMZ	(if Infinity×Zero)
VXSQRT	(if Invalid Square Root)
VXCVI	(if Invalid Integer Convert)

  2. Update of VSR[XT] is suppressed.
  3. FR and FI are set to zero.
  4. FPRF is unchanged.
- For *VSX Scalar Floating-Point Compare* instructions:
  1. One or two of the following Invalid Operation exceptions are set to 1.

VXSNAN	(if SNaN)
VXVC	(if Invalid Compare)

  2. FR, FI, and C are unchanged.
  3. FPCC is set to reflect unordered.
- For any of the following instructions,
  VSX Scalar Quad-Precision Arithmetic instructions:
  xsaddqp[***o***], ***xsdivqp***[***o***], ***xsmulqp***[***o***], ***xssqrtqp***[***o***], ***xssubqp***[***o***] ***xsmaddqp***[***o***], ***xsmsubqp***[***o***], ***xsnmaddqp***[***o***], ***xsnmsubqp***[***o***]
  VSX Scalar Quad-Precision Convert to Integer instructions:
  xscvqpsdz, ***xscvqpswz***, ***xscvqpudz***, ***xscvqpuwz***
  VSX Scalar Round Quad-Precision to Double-Extended-Precision (***xsrqpxp***)
  VSX Scalar Round to Quad-Precision Integer (***xsrqpi***)
  VSX Scalar Round to Quad-Precision Integer with Inexact (***xsrqpix***)
  VSX Scalar Convert with round Quad-Precision to Double-Precision format [*using round to Odd*] (***xscvqpdp***[***o***])

do the following.

1. One or two of the following Invalid Operation exceptions are set to 1.

VXSNAN	(if SNaN)
VXISI	(if Infinity—Infinity)
VXIDI	(if Infinity÷Infinity)
VXZDZ	(if Zero÷Zero)
VXIMZ	(if Infinity×Zero)
VXSQRT	(if Invalid Square Root)
VXCVI	(if Invalid Integer Convert)

2. VSR[VRT+32] is not modified.
3. FR and FI are set to zero. FPRF is not modified.

- For any of the following instructions,
  VSX Scalar Compare Ordered Quad-Precision (***xscmpoqp***)
  VSX Scalar Compare Unordered Quad-Precision (***xscmpuqp***)

  do the following.
  1. One or two of the following Invalid Operation exceptions are set to 1.

VXSNAN	(if SNaN)
VXVC	(if Invalid Compare)

  2. FR, FI, and C are not modified. FPCC is set to reflect unordered.
- For any of the following instructions,
  VSX Scalar Convert Half-Precision to Double-Precision format (***xscvhpdp***)
  VSX Scalar Convert with round Double-Precision to Half-Precision format (***xscvdphp***)

  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 (***xvcvhpsp***)
  VSX Vector Convert with round Single-Precision to Half-Precision format (***xvcvsphp***)
  VSX Vector Convert with round Single-Precision to bfloat16 format (***xvcvspbf16***)

  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 elements 0 and 1 of VSR[XT]. The contents of word elements 2 and 3 of VSR[XT] are set to 0.
  3. FR and FI are set to 0.
  4. FPRF is set to indicate the class of the result (Quiet NaN).
- For the *VSX Vector Single-Precision Arithmetic* instructions, *VSX Vector Single-Precision Maximum/Minimum* instructions, the *VSX Vector Convert with round Double-Precision to Single-Precision format* (***xvcvdpsp***) instruction, and the *VSX Vector Round to Single-Precision Integer* instructions:
  1. One or two of the following Invalid Operation exceptions are set to 1.

VXSNAN	(if SNaN)
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], and for ***xvcvdpsp***, is also placed into bits 32:63 of its respective doubleword element of VSR[XT].
  3. FR, FI, and FPRF are not modified.
- For the *VSX Scalar Double-Precision Arithmetic* instructions, *VSX Scalar Double-Precision Maximum/Minimum* instructions, the *VSX Scalar Convert Single-Precision to Double-Precision format* (***xscvspdp***) instruction, and the *VSX Scalar Round to Double-Precision Integer* instructions:
  1. One or two of the following Invalid Operation exceptions are set to 1.

VXSNAN	(if SNaN)
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 set to 0.
  3. FR and FI are set to 0.
  4. FPRF is set to indicate the class of the result (Quiet NaN).
- For any of the following instructions,

  VSX Scalar Quad-Precision Arithmetic instructions:
  xsaddqp[***o***], ***xsdivqp***[***o***], ***xsmulqp***[***o***], ***xssqrtqp***[***o***], ***xssubqp***[***o***] ***xsmaddqp***[***o***], ***xsmsubqp***[***o***], ***xsnmaddqp***[***o***], ***xsnmsubqp***[***o***]
  VSX Scalar Quad-Precision Round to Integer (***xsrqpi***)
  VSX Scalar Quad-Precision Round to Integer with Inexact (***xsrqpix***)

  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* (***xsrqpxp***), do the following.
  1. VXSNAN is set to 1.
  2. The Quiet NaN is placed into VSR[VRT+32] in quad-precision format.

3. FR and FI are set to 0. FPRF is set to indicate the class of the result (Quiet NaN).

- For any of the following instructions,

  VSX Scalar Compare Ordered Quad-Precision (***xscmpoqp***)
  VSX Scalar Compare Unordered Quad-Precision (***xscmpoqp***)

  do the following.

  1. One or two of the following Invalid Operation exceptions are set to 1.

VXSNAN	(if SNaN)
VXVC	(if Invalid Compare)

  2. FR, FI and C are unchanged. FPCC is set to reflect unordered.

- For *VSX Scalar Convert with round Quad-Precision to Double-Precision format* [using round *to Odd*] (***xscvqpdp***[***o***]), do the following.

  1. VXSNAN is set to 1.
  2. The double-precision Quiet NaN result is placed into doubleword element 0 of VSR[VRT+32] in double-precision format.

     0x0000_0000_0000_0000 is placed into doubleword element 1 of VSR[VRT+32].
  3. FR and FI are set to 0. FPRF is set to indicate the class of the result (Quiet NaN).

- For *VSX Scalar Convert with round to zero Quad-Precision to Signed Doubleword format* (***xscvqpsdz***), do the following.

  1. One or two of the following Invalid Operation exceptions are set to 1.

VXSNAN	(if SNaN)
VXCVI	(if Invalid Integer Convert)

  2. 0x7FFF_FFFF_FFFF_FFFF is placed into doubleword element 0 of VSR[VRT+32] if the quad-precision operand in VSR[VRB+32] is a positive number or +Infinity.

     0x8000_0000_0000_0000 is placed into doubleword element 0 of VSR[VRT+32] if the quad-precision operand in VSR[VRB+32] is a negative number, −Infinity, or NaN.

     0x0000_0000_0000_0000 is placed into doubleword element 1 of VSR[VRT+32].
  3. FR and FI are set to 0. FPRF is undefined.

- For *VSX Scalar Convert with round to zero Quad-Precision to Signed Word format* (***xscvqpswz***), do the following.

  1. One or two of the following Invalid Operation exceptions are set to 1.

VXSNAN	(if SNaN)
VXCVI	(if Invalid Integer Convert)

  2. 0x7FFF_FFFF is placed into word element 1 of VSR[VRT+32] if the quad-precision operand in VSR[VRB+32] is a positive number or +Infinity.

     0x8000_0000 is placed into word element 1 of VSR[VRT+32] if the quad-precision operand in VSR[VRB+32] is a negative number, −Infinity, or NaN.

     0x0000_0000 is placed into word elements 0, 2, and 3 of VSR[VRT+32].
  3. FR and FI are set to 0. FPRF is undefined.

- For *VSX Scalar Convert with round to zero Quad-Precision to Unsigned Doubleword format* (***xscvqpudz***), do the following.

  1. One or two of the following Invalid Operation exceptions are set to 1.

VXSNAN	(if SNaN)
VXCVI	(if Invalid Integer Convert)

  2. 0xFFFF_FFFF_FFFF_FFFF is placed into doubleword element 0 of VSR[VRT+32] if the quad-precision operand in VSR[VRB+32] is a positive number or +Infinity.

     0x0000_0000_0000_0000 is placed into doubleword element 0 of VSR[VRT+32] if the quad-precision operand in VSR[VRB+32] is a negative number, −Infinity, or NaN.

     0x0000_0000_0000_0000 is placed into doubleword element 1 of VSR[VRT+32].
  3. FR and FI are set to 0. FPRF is undefined.

- For *VSX Scalar Convert with round to zero Quad-Precision to Unsigned Word format* (***xscvqpuwz***), do the following.

  1. One or two of the following Invalid Operation exceptions are set to 1.

VXSNAN	(if SNaN)
VXCVI	(if Invalid Integer Convert)

2. 0xFFFF_FFFF is placed into word element 1 of VSR[VRT+32] if the quad-precision operand in VSR[VRB+32] is a positive number or +Infinity.

   0x0000_0000 is placed into word element 1 of VSR[VRT+32] if the quad-precision operand in VSR[VRB+32] is a negative number, −Infinity, or NaN.

   0x0000_0000 is placed into word elements 0, 2, and 3 of VSR[VRT+32].
3. FR and FI are set to 0. FPRF is undefined.

- For *VSX Scalar Convert with round Double-Precision to Half-Precision format* (***xscvdphp***), 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 doubleword element 0 of VSR[XT]. The contents of the leftmost 3 halfwords of doubleword element 0 of VSR[XT] are set to 0. The contents of doubleword element 1 of VSR[XT] are set to 0.
  3. FR and FI are set to 0. FPRF is set to indicate the class of the result (Quiet NaN).
- For *VSX Scalar Convert Half-Precision to Double-Precision format* (***xscvhpdp***), 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 set to 0.
  3. FR and FI are set to 0. FPRF is set to indicate the class of the result (Quiet NaN).
- For any of the following instructions,

  VSX Vector Double-Precision Arithmetic instructions
  VSX Vector Double-Precision Maximum/Minimum instructions
  VSX Vector Convert Single-Precision to Double-Precision format (***xvcvspdp***)
  VSX Vector Round to Double-Precision Integer instructions

  do the following.
  1. One or two of the following Invalid Operation exceptions are set to 1.

VXSNAN	(if SNaN)
VXISI	(if Infinity–Infinity)
VXIDI	(if Infinity÷Infinity)
VXZDZ	(if Zero÷Zero)
VXIMZ	(if Infinity×Zero)
VXSQRT	(if Invalid Square Root)

  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, do the following.
  1. One or two of the following Invalid Operation exceptions are set to 1.

VXSNAN	(if SNaN)
VXCVI	(if Invalid Integer Convert)

  2. 0x7FFF_FFFF_FFFF_FFFF is placed into doubleword element 0 of VSR[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 set to 0.
  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, do the following.
  1. One or two of the following Invalid Operation exceptions are set to 1.

VXSNAN	(if SNaN)
VXCVI	(if Invalid Integer Convert)

  2. 0xFFFF_FFFF_FFFF_FFFF is placed into doubleword element 0 of VSR[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 set to 0.

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* (***xscvdpsxw***) instruction, do the following.
  1. One or two of the following Invalid Operation exceptions are set to 1.

     VXSNAN (if SNaN)
     VXCVI (if Invalid Integer Convert)
  2. 0x7FFF_FFFF is placed into word elements 0 and 1 of VSR[XT] if the double-precision operand in doubleword element 0 of VSR[XB] is a positive number or +Infinity.

     0x8000_0000 is placed into word elements 0 and 1 of VSR[XT] if the double-precision operand in doubleword element 0 of VSR[XB] is a negative number, −Infinity, or NaN.

     The contents of word elements 2 and 3 of VSR[XT] are set to 0.
  3. FR and FI are set to 0.
  4. FPRF is undefined.
- For the *VSX Scalar Convert with round to zero Double-Precision to Unsigned Word format* (***xscvdpuxw***) instruction, do the following.
  1. One or two of the following Invalid Operation exceptions are set to 1.

     VXSNAN (if SNaN)
     VXCVI (if Invalid Integer Convert)
  2. 0xFFFF_FFFF is placed into word elements 0 and 1 of VSR[XT] if the double-precision operand in doubleword element 0 of VSR[XB] is a positive number or +Infinity.

     0x0000_0000 is placed into word elements 0 and 1 of VSR[XT] if the double-precision operand in doubleword element 0 of VSR[XB] is a negative number, −Infinity, or NaN.

     The contents of word elements 2 and 3 of VSR[XT] are set to 0.
  3. FR and FI are set to 0.
  4. FPRF is undefined.
- For the *VSX Vector Convert with round to zero Double-Precision to Signed Doubleword format* (***xvcvdpsxd***) instruction, do the following.
  1. One or two of the following Invalid Operation exceptions are set to 1.

     VXSNAN (if SNaN)
     VXCVI (if Invalid Integer Convert)
  2. 0x7FFF_FFFF_FFFF_FFFF is placed into doubleword element i of VSR[XT] if the double-precision operand in the corresponding doubleword element of VSR[XB] is a positive number or +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, −Infinity, or NaN.
  3. FR, FI, and FPRF are not modified.
- For the *VSX Vector Convert with round to zero Double-Precision to Unsigned Doubleword format* (***xvcvdpuxd***) instruction, do the following.
  1. One or two of the following Invalid Operation exceptions are set to 1.

     VXSNAN (if SNaN)
     VXCVI (if Invalid Integer Convert)
  2. 0xFFFF_FFFF_FFFF_FFFF is placed into doubleword element i of VSR[XT] if the double-precision operand in doubleword element i of VSR[XB] is a positive number or +Infinity.

     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, −Infinity, 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* (***xvcvdpsxw***) instruction, do the following.
  1. One or two of the following Invalid Operation exceptions are set to 1.

     VXSNAN (if SNaN)
     VXCVI (if Invalid Integer Convert)
  2. 0x7FFF_FFFF is placed into word elements i×2 and i×2+1 of VSR[XT] if the double-precision operand in doubleword element i of VSR[XB] is a positive number or +Infinity.

     0x8000_0000 is placed into word elements i×2 and i×2+1 of VSR[XT] if the double-precision operand in doubleword element i of VSR[XB] is a negative number, −Infinity, or NaN.

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, do the following.
  1. One or two of the following Invalid Operation exceptions are set to 1.

VXSNAN	(if SNaN)
VXCVI	(if Invalid Integer Convert)

  2. 0xFFFF_FFFF is placed into word elements i×2 and i×2+1 of VSR[XT] if the double-precision operand in doubleword element i of VSR[XB] is a positive number or +Infinity.

     0x0000_0000 is placed into word elements i×2 and i×2+1 of VSR[XT] if the double-precision operand in doubleword element i of VSR[XB] is a negative number, −Infinity, or NaN.
  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, do the following.
  1. One or two of the following Invalid Operation exceptions are set to 1.

VXSNAN	(if SNaN)
VXCVI	(if Invalid Integer Convert)

  2. 0x7FFF_FFFF_FFFF_FFFF is placed into doubleword element i of VSR[XT] if the single-precision operand in word element i×2 of VSR[XB] is a positive number or +Infinity.

     0x8000_0000_0000_0000 is placed into doubleword element i of VSR[XT] if the single-precision operand in word element i×2 of VSR[XB] is a negative number, −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, do the following.
  1. One or two of the following Invalid Operation exceptions are set to 1.

VXSNAN	(if SNaN)
VXCVI	(if Invalid Integer Convert)

  2. 0xFFFF_FFFF_FFFF_FFFF is placed into doubleword element i of VSR[XT] if the single-precision operand in word element i×2 of VSR[XB] is a positive number or +Infinity.

     0x0000_0000_0000_0000 is placed into doubleword element i of VSR[XT] if the single-precision operand in word element i×2 of VSR[XB] is a negative number, −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, do the following.
  1. One or two of the following Invalid Operation exceptions are set to 1.

VXSNAN	(if SNaN)
VXCVI	(if Invalid Integer Convert)

  2. 0x7FFF_FFFF is placed into word element i of VSR[XT] if the single-precision operand in word element i of VSR[XB] is a positive number or +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, −Infinity, or NaN.
  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, do the following.
  1. One or two of the following Invalid Operation exceptions are set to 1.

VXSNAN	(if SNaN)
VXCVI	(if Invalid Integer Convert)

  2. 0xFFFF_FFFF is placed into word element i of VSR[XT] if the single-precision operand in the corresponding word element 2×i of VSR[XB] is a positive number or +Infinity.

     0x0000_0000 is placed into word element i of VSR[XT] if the single-precision operand in word element 2×i of VSR[XB] is a negative number, −Infinity, or NaN.
  3. FR, FI, and FPRF are not modified.
- For the *VSX Scalar Floating-Point Compare* instructions, do the following.
  1. One or two of the following Invalid Operation exceptions are set to 1.

VXSNAN	(if SNaN)
VXVC	(if Invalid Compare)

2. FR, FI and C are unchanged.
3. FPCC is set to reflect unordered.

- For the VSX *Vector Compare Single-Precision* instructions, do the following.
  1. One or two of the following Invalid Operation exceptions are set to 1.

     VXSNAN (if SNaN)
     VXVC (if Invalid Compare)

  2. 0x0000_0000 is placed into its respective word element of VSR[XT].
  3. FR, FI, and FPRF are not modified.
- For the *Vector Double-Precision Compare* instructions, do the following.
  1. One or two of the following Invalid Operation exceptions are set to 1.

     VXSNAN (if SNaN)
     VXVC (if Invalid Compare)

  2. 0x0000_0000_0000_0000 is placed into its respective doubleword element of VSR[XT].
  3. FR, FI, and FPRF are not modified.
- For any of the following instructions,

  VSX Vector Convert with round Single-Precision to Half-Precision format (***xvcvsphp***)
  VSX Vector Convert with round Single-Precision to bfloat16 format (***xvcvspbf16***)

  do the following.
  1. VXSNAN is set to 1.
  2. The half-precision representation of a Quiet NaN is placed into the rightmost halfword of its respective word element of VSR[XT]. The contents of the leftmost halfword of its respective word element of VSR[XT] are set to 0.
  3. FR, FI, and FPRF are not modified.
- For *VSX Vector Convert Half-Precision to Single-Precision format* (***xvcvhpsp***), do the following.
  1. VXSNAN is set to 1.
  2. The half-precision representation of a Quiet NaN is placed into the respective word element of VSR[XT].
  3. FR, FI, and FPRF are not modified.

7.4.2 Floating-Point Zero Divide Exception

7.4.2.1 Definition

A Zero Divide exception occurs when a *VSX Floating-Point Divide* instruction is executed with a zero divisor value and a finite nonzero dividend value.

A Zero Divide exception also occurs when a *VSX Floating-Point Reciprocal Estimate* instruction or a *VSX Floating-Point Reciprocal Square Root Estimate* instruction is executed with an operand value of zero.

The action to be taken depends on the setting of the Zero Divide Exception Enable bit of the FPSCR.

7.4.2.2 Action for ZE=1

When Zero Divide exception is enabled (`ZE=1`) and a Zero Divide exception occurs, the following actions are taken:

- For any of the following instructions,
  VSX Scalar Divide Double-Precision (***xsdivdp***)
  VSX Scalar Divide Single-Precision (***xsdivsp***)
  VSX Scalar Divide Quad-Precision (***xsdivqp***)
  VSX Scalar Reciprocal Estimate Double-Precision (***xsredp***)
  VSX Scalar Reciprocal Estimate Single-Precision (***xsresp***)
  VSX Scalar Reciprocal Square Root Estimate Double-Precision (***xsrsqrtedp***)
  VSX Scalar Reciprocal Square Root Estimate Single-Precision (***xsrsqrtesp***)

  do the following.
  1. `ZX` is set to 1.
  2. `VSR[XT]` is not modified.
  3. `FR` and `FI` are set to 0. `FPRF` is unchanged.
- For any of the following instructions,
  VSX Vector Divide Double-Precision (***xvdivdp***)
  VSX Vector Divide Single-Precision (***xvdivsp***)
  VSX Vector Reciprocal Estimate Double-Precision (***xvredp***)
  VSX Vector Reciprocal Estimate Single-Precision (***xvresp***)
  VSX Vector Reciprocal Square Root Estimate Double-Precision (***xvrsqrtedp***)
  VSX Vector Reciprocal Square Root Estimate Single-Precision (***xvrsqrtesp***)

  do the following.
  1. `ZX` is set to 1.
  2. `VSR[XT]` is not modified.
  3. `FR` and `FI` are unchanged. `FPRF` is unchanged.

7.4.2.3 Action for ZE=0

When Zero Divide exception is disabled (`ZE=0`) and a Zero Divide exception occurs, the following actions are taken:

- For any of the following instructions,
  VSX Scalar Divide Double-Precision (***xsdivdp***)
  VSX Scalar Divide Single-Precision (***xsdivsp***)

  do the following.
  1. `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 set to 0.
  3. `FR` and `FI` are set to 0. `FPRF` is set to indicate the class and sign of the result ($\pm$ Infinity).
- For *VSX Scalar Divide Quad-Precision* (***xsdivqp***), 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* (***xvdivdp***), 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* (***xvdivsp***), 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.
- For any of the following instructions,

  VSX Scalar Reciprocal Estimate Double-Precision (***xsredp***)
  VSX Scalar Reciprocal Estimate Single-Precision (***xsresp***)
  VSX Scalar Reciprocal Square Root Estimate Double-Precision (***xsrsqrtedp***)
  VSX Scalar Reciprocal Square Root Estimate Single-Precision (***xsrsqrtesp***)

  do the following.
  1. ZX is set to 1.
  2. An Infinity, having the sign of the source operand, is placed into doubleword element 0 of VSR[XT] in double-precision format. The contents of doubleword element 1 of VSR[XT] are set to 0.
  3. FR and FI are set to 0. FPRF is set to indicate the class and sign of the result ($\pm$ Infinity).
- For any of the following instructions,

  VSX Vector Reciprocal Estimate Double-Precision (***xsredp***)
  VSX Vector Reciprocal Square Root Estimate Double-Precision (***xsrsqrtedp***)

  do the following.
  1. ZX is set to 1.
  2. For each vector element causing a Zero Divide exception, an Infinity, having the sign of the source operand, is placed into its respective doubleword element of VSR[XT] in double-precision format.
  3. FR, FI, and FPRF are not modified.
- For any of the following instructions,

  VSX Vector Reciprocal Estimate Single-Precision (***xsresp***)
  VSX Vector Reciprocal Square Root Estimate Single-Precision (***xsrsqrtesp***)

  do the following.
  1. ZX is set to 1.
  2. For each vector element causing a Zero Divide exception, an Infinity, having the sign of the source operand, is placed into its respective word element of VSR[XT] in single-precision format.
  3. FR, FI, and FPRF are not modified.

7.4.3 Floating-Point Overflow Exception

7.4.3.1 Definition

An Overflow exception occurs when the magnitude of what would have been the rounded result if the exponent range were unbounded exceeds that of the largest finite number of the specified result precision.

The action to be taken depends on the setting of the Overflow Exception Enable bit of the FPSCR.

7.4.3.2 Action for OE=1

When Overflow exception is enabled (`OE=1`) and an Overflow exception occurs, the following actions are taken:

- For the *VSX Vector round and Convert Double-Precision to Single-Precision format* (***xscvdpsp***) instruction:
  1. `OX` is set to 1.
  2. If the unbiased exponent of the normalized intermediate result is less than or equal to 318 (`Emax + 192`), the exponent is adjusted by subtracting 192. Otherwise the result is undefined.
  3. The adjusted rounded result is placed into word elements 0 and 1 of `VSR[XT]` in single-precision format. The contents of word elements 2 and 3 of `VSR[XT]` are set to 0.
  4. Unless the result is undefined, `FPRF` is set to indicate the class and sign of the result (±Normal Number).
- For any of the following instructions,

  VSX Scalar Double-Precision Arithmetic instructions
  xsadddp, ***xsdivdp***, ***xsmuldp***, ***xssubdp***, ***xsmaddadp***, ***xsmaddmdp***, ***xsmsubadp***, ***xsmsubmdp***, ***xsnmaddadp***, ***xsnmaddmdp***, ***xsnmsubadp***, ***xsnmsubmdp***
  VSX Vector Reciprocal Estimate Double-Precision (***xsredp***)

  do the following.
  1. `OX` is set to 1.
  2. The exponent of the normalized intermediate result is adjusted by subtracting 1536.
  3. The adjusted rounded result is placed into doubleword element 0 of `VSR[XT]` in double-precision format. The contents of doubleword element 1 of `VSR[XT]` are set to 0.
  4. `FPRF` is set to indicate the class and sign of the result (±Normal Number).
- For any of the following instructions,

  VSX Scalar Single-Precision Arithmetic instructions
  xsaddsp, ***xsdivsp***, ***xsmulsp***, ***xssubsp xsmaddasp***, ***xsmaddmsp***, ***xsmsubasp***, ***xsmsubmsp xsnmaddasp***, ***xsnmaddmsp***, ***xsnmsubasp***, ***xsnmsubmsp***
  VSX Scalar Reciprocal Estimate Single-Precision (***xsresp***)
  VSX Vector Reciprocal Square Root Estimate Single-Precision (***xsrsqrtesp***)
  VSX Scalar Round to Single-Precision (***xsrsp***)

  do the following.
  1. `OX` is set to 1.
  2. The exponent is adjusted by subtracting 192.
  3. The adjusted and rounded result is placed into doubleword element 0 of `VSR[XT]` in double-precision format. The contents of doubleword element 1 of `VSR[XT]` are set to 0.
  4. `FPRF` is set to indicate the class and sign of the result (±Normal Number).
- For any of the following instructions,

  VSX Scalar Quad-Precision Arithmetic instructions
  xsaddqp[***o***], ***xsdivqp***[***o***], ***xsmulqp***[***o***], ***xssqrtqp***[***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. `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*] (***xscvqpdp***), 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 (Emax), 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 (Emax), the result is undefined.
  3. The adjusted, rounded result is placed into rightmost halfword of doubleword element 0 of VSR[XT] in half-precision format.
     The contents of the leftmost 3 halfwords of doubleword element 0 of VSR[XT] are set to 0.
     The contents of doubleword element 1 of VSR[XT] are set to 0.
  4. Unless the result is undefined, FPRF is set to indicate the class and sign of the result (±Normal Number).
- For any of the following instructions,
  VSX Vector Double-Precision Arithmetic instructions
  xvadddp, ***xvdivdp***, ***xvmuldp***, ***xvredp***, ***xvsubdp***, ***xvmaddadp***, ***xsmaddmdp***, ***xvmsubadp***, ***xvmsubmdp***, ***xvnmaddadp***, ***xvnmaddmdp***, ***xvnmsubadp***, ***xvnmsubmdp***
  VSX Vector Single-Precision Arithmetic instructions
  xvaddsp, ***xvdivsp***, ***xvmulsp***, ***xvresp***, ***xvsubsp***, ***xvmaddasp***, ***xvmaddmsp***, ***xvsmsubasp***, ***xvmsubmsp***, ***xvnmaddasp***, ***xvnmaddmsp***, ***xvnmsubasp***, ***xvnmsubmsp***
  VSX Vector round and Convert Double-Precision to Single-Precision format (***xvcvdpsp***)
  do the following.
  1. OX is set to 1.
  2. VSR[XT] is not modified.
  3. FR, FI, and FPRF are not modified.
- For any of the following instructions,
  VSX Vector Convert with round Single-Precision to Half-Precision format (***xvcvsphp***)
  VSX Vector Convert with round Single-Precision to bfloat16 format (***xvcvspbf16***)
  do the following.
  1. OX is set to 1.
  2. VSR[XT] is not modified.
  3. FR, FI, and FPRF are not modified.

7.4.3.3 Action for OE=0

When Overflow exception is disabled (OE=0) and an Overflow exception occurs, the following actions are taken:

- For all instructions,
  1. OX and XX are set to 1.
  2. The result is determined by the rounding mode (RN) and the sign of the intermediate result as follows:
     Round to Nearest Even
     For negative overflow, the result is −Infinity.
     For positive overflow, the result is +Infinity.
     Round toward Zero
     For negative overflow, the result is the format's most negative finite number.
     For positive overflow, the result is the format's most positive finite number.
     Round toward +Infinity
     For negative overflow, the result is the format's most negative finite number.
     For positive overflow, the result is +Infinity.
     Round toward −Infinity
     For negative overflow, the result is −Infinity.
     For positive overflow, the result is the format's most positive finite number.
- For *VSX Scalar round and Convert Double-Precision to Single-Precision format* (***xscvdpsp***):

3. The result is placed into word elements 0 and 1 of VSR[XT] as a single-precision value. The contents of word elements 2 and 3 of VSR[XT] are set to 0.
4. FR is undefined. FI is set to 1. FPRF is set to indicate the class and sign of the result.

- For any of the following instructions,
  VSX Scalar Double-Precision Arithmetic instructions
  xsadddp, ***xsdivdp***, ***xsmuldp***, ***xsredp***, ***xssubdp***, ***xsmaddadp***, ***xsmaddmdp***, ***xsmsubadp***, ***xsmsubmdp***, ***xsnmaddadp***, ***xsnmaddmdp***, ***xsnmsubadp***, ***xsnmsubmdp***
  VSX Scalar Single-Precision Arithmetic instructions
  xsaddsp, ***xsdivsp***, ***xsmulsp***, ***xsresp***, ***xssubsp***, ***xsmaddasp***, ***xsmaddmsp***, ***xsmsubasp***, ***xsmsubmsp***, ***xsnmaddasp***, ***xsnmaddmsp***, ***xsnmsubasp***, ***xsnmsubmsp***
  do the following.
  3. The result is placed into doubleword element 0 of VSR[XT] as a double-precision value. The contents of doubleword element 1 of VSR[XT] are set to 0.
  4. FR is undefined. FI is set to 1. FPRF is set to indicate the class and sign of the result.
- For any of the following instructions,
  VSX Scalar Quad-Precision Arithmetic instructions
  xsaddqp[***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***)
  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* (***xscvqpdp***), do the following.
  3. The result is placed into doubleword element 0 of VSR[VRT+32] as a double-precision value.
     0x0000_0000_0000_0000 is placed into doubleword element 1 of VSR[VRT+32].
  4. FR is undefined. FI is set to 1. FPRF is set to indicate the class and sign of the result.
- For *VSX Scalar Convert with round Double-Precision to Half-Precision format* (***xscvdphp***), do the following.
  3. The result is placed into the rightmost halfword of doubleword element 0 of VSR[XT] as a half-precision value.
     The contents of the leftmost 3 halfwords of doubleword element 0 of VSR[XT] are set to 0.
     The contents of doubleword element 1 of VSR[XT] are set to 0.
  4. FR is undefined. FI is set to 1. FPRF is set to indicate the class and sign of the result.
- For any of the following instructions,
  VSX Vector Double-Precision Arithmetic instructions
  xvadddp, ***xvdivdp***, ***xvmuldp***, ***xvredp***, ***xvsubdp***, ***xvmaddadp***, ***xvmaddmdp***, ***xvmsubadp***, ***xvmsubmdp***, ***xvnmaddadp***, ***xvnmaddmdp***, ***xvnmsubadp***, ***xvnmsubmdp***
  do the following.
  3. For each vector element causing an Overflow exception, the result is placed into its respective doubleword element of VSR[XT] in double-precision format.
  4. FR, FI, and FPRF are not modified.
- For any of the following instructions,
  VSX Vector Single-Precision Arithmetic instructions
  xvaddsp, ***xvdivsp***, ***xvmulsp***, ***xvresp***, ***xvsubsp***, ***xvmaddasp***, ***xvmaddmsp***, ***xvmsubasp***, ***xvmsubmsp***, ***xvnmaddasp***, ***xvnmaddmsp***, ***xvnmsubasp***, ***xvnmsubmsp***
  VSX Vector round and Convert Double-Precision to Single-Precision format (***xvcvdpsp***)
  do the following.
  3. For each vector element causing an Overflow exception, the result is placed into its respective word element of VSR[XT] in single-precision format, and for ***xvcvdpsp***, is also placed into bits 32:63 of its respective doubleword element of VSX[XT].
  4. FR, FI, and FPRF are not modified.
- For any of the following instructions,
  VSX Vector Convert with round Single-Precision to Half-Precision format (***xvcvsphp***)
  VSX Vector Convert with round Single-Precision to bfloat16 format (***xvcvspbf16***)
  do the following.
  3. For each vector element causing an Overflow exception, the result is placed into the rightmost halfword of its respective word element of VSR[XT] in half-precision format.
     The contents of the leftmost halfword of its respective word element of VSR[XT] are set to 0.
  4. FR, FI, and FPRF are not modified.

7.4.4 Floating-Point Underflow Exception

7.4.4.1 Definition

Underflow exception is defined separately for the enabled and disabled states:

Enabled:
Underflow occurs when the intermediate result is “Tiny”.

Disabled:
Underflow occurs when the intermediate result is “Tiny” and there is “Loss of Accuracy”.

A *tiny* result is detected before rounding, when a nonzero intermediate result computed as though both the precision and the exponent range were unbounded would be less in magnitude than the smallest normalized number.

If the intermediate result is tiny and Underflow exception is disabled (`UE=0`), the intermediate result is denormalized (see Section 7.3.2.4, “Normalization and Denormalization” on page 495) and rounded (see Section 7.3.2.6, “Rounding” on page 499) 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* (***xscvdpsp***), do the following.
  1. `UX` is set to 1.
  2. The exponent of the normalized intermediate result is adjusted by adding `192`. If the adjusted unbiased exponent is less than `-126` (`Emin`), the result is undefined.
  3. The adjusted rounded result is placed into word elements 0 and 1 of `VSR[XT]` in single-precision format. The contents of word elements 2 and 3 of `VSR[XT]` are undefined.
  4. Unless the result is undefined, `FPRF` is set to indicate the class and sign of the result (±Normal Number).
- For any of the following instructions,
  VSX Scalar Double-Precision Arithmetic instructions
  xsadddp, ***xsdivdp***, ***xsmuldp***, ***xssubdp***, ***xsmaddadp***, ***xsmaddmdp***, ***xsmsubadp***, ***xsmsubmdp***, ***xsnmaddadp***, ***xsnmaddmdp***, ***xsnmsubadp***, ***xsnmsubmdp***
  VSX Scalar Double-Precision Reciprocal Estimate (***xsredp***)
  do the following.
  1. `UX` is set to 1.
  2. The exponent of the normalized intermediate result is adjusted by adding 1536.
  3. The adjusted rounded result is placed into word elements 0 and 1 of `VSR[XT]` in single-precision format. The contents of word elements 2 and 3 of `VSR[XT]` are set to 0
  4. `FPRF` is set to indicate the class and sign of the result (±Normal Number).
- For any of the following instructions,
  VSX Scalar Quad-Precision Arithmetic instructions
  xsaddqp[***o***], ***xsdivqp***[***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 exponent of the normalized intermediate result is adjusted by adding `24576`.
  3. The adjusted, rounded result is placed into `VSR[VRT+32]` in quad-precision format.
  4. `FPRF` is set to indicate the class and sign of the result (±Normal Number).
- For *VSX Scalar Convert with round Quad-Precision to Double-Precision format* [*using round to Odd*] (***xscvqpdp***[***o***]), do the following.
  1. `UX` is set to 1.

2. The exponent of the normalized intermediate result is adjusted by adding 1536. If the adjusted unbiased exponent is less than −1022 (Emin), 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 any of the following instructions,

  VSX Scalar Single-Precision Arithmetic instructions
  xsaddsp, ***xsdivsp***, ***xsmulsp***, ***xssubsp***, ***xsmaddasp***, ***xsmaddmsp***, ***xsmsubasp***, ***xsmsubmsp***, ***xsnmaddasp***, ***xsnmaddmsp***, ***xsnmsubasp***, ***xsnmsubmsp***
  VSX Scalar Single-Precision Reciprocal Estimate (***xsresp***)

  do the following.

  1. UX is set to 1.
  2. The exponent of the normalized intermediate result is adjusted by adding 192. If the adjusted unbiased exponent is less than −126 (Emin), the result is undefined.
  3. The adjusted rounded result is placed into doubleword element 0 of VSR[XT] in double-precision format. The contents of doubleword element 1 of VSR[XT] are set to 0.
  4. Unless the result is undefined, 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.
  2. The exponent of the normalized intermediate result is adjusted by adding 24. If the adjusted unbiased exponent is less than −14, the result is undefined.
  3. The adjusted, rounded result is placed into rightmost halfword of doubleword element 0 of VSR[XT] in half-precision format.

     The contents of the leftmost 3 halfwords of doubleword element 0 of VSR[XT] are set to 0.

     The contents of doubleword element 1 of VSR[XT] are set to 0.
  4. Unless the result is undefined, FPRF is set to indicate the class and sign of the result (±Normal Number).
- For any of the following instructions,

  VSX Vector Double-Precision Arithmetic instructions
  xvadddp, ***xvdivdp***, ***xvmuldp***, ***xvsubdp***, ***xvmaddadp***, ***xvmaddmdp***, ***xvmsubadp***, ***xvmsubmdp***, ***xvnmaddadp***, ***xvnmaddmdp***, ***xvnmsubadp***, ***xvnmsubmdp***
  VSX Vector Single-Precision Arithmetic instructions
  xvaddsp, ***xvdivsp***, ***xvmulsp***, ***xvsubsp***, ***xvmaddasp***, ***xvmaddmsp***, ***xvmsubasp***, ***xvmsubmsp***, ***xvnmaddasp***, ***xvnmaddmsp***, ***xvnmsubasp***, ***xvnmsubmsp***
  VSX Vector Reciprocal Estimate Double-Precision (***xvredp***)
  VSX Vector Reciprocal Estimate Single-Precision (***xvresp***)
  VSX Vector round and Convert Double-Precision to Single-Precision format (***xvcvdpsp***)
  VSX Vector Convert with round Single-Precision to Half-Precision format (***xvcvsphp***)
  VSX Vector Convert with round Single-Precision to bfloat16 format (***xvcvspbf16***)

  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 elements 0 and 1 of VSR[XT] as a single-precision value. The contents of word elements 2 and 3 of VSR[XT] are set to 0.
3. FPRF is set to indicate the class and sign of the result.

- For any of the following instructions,
  VSX Scalar Double-Precision Arithmetic instructions
  xsadddp, ***xsdivdp***, ***xsmuldp***, ***xssubdp***, ***xsmaddadp***, ***xsmaddmdp***, ***xsmsubadp***, ***xsmsubmdp***, ***xsnmaddadp***, ***xsnmaddmdp***, ***xsnmsubadp***, ***xsnmsubmdp***
  VSX Scalar Single-Precision Arithmetic instructions
  xsaddsp, ***xsdivsp***, ***xsmulsp***, ***xssubsp***, ***xsmaddasp***, ***xsmaddmsp***, ***xsmsubasp***, ***xsmsubmsp***, ***xsnmaddasp***, ***xsnmaddmsp***, ***xsnmsubasp***, ***xsnmsubmsp***
  VSX Scalar Reciprocal Estimate Double-Precision (***xsredp***)
  VSX Scalar Reciprocal Estimate Single-Precision (***xsresp***)
  do the following.
  1. UX is set to 1.
  2. The result is placed into doubleword element 0 of VSR[XT] in double-precision format. The contents of doubleword element 1 of VSR[XT] are set to 0.
  3. FPRF is set to indicate the class and sign of the result.
- For any of the following instructions,
  VSX Scalar Quad-Precision Arithmetic instructions
  xsaddqp[***o***], ***xsdivqp***[***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* (***xscvqpdp***), 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 set to 0.
  3. FPRF is set to indicate the class and sign of the result.
- For any of the following instructions,
  VSX Vector Double-Precision Arithmetic instructions
  xvadddp, ***xvdivdp***, ***xvmuldp***, ***xvsubdp*** ***xvmaddadp***, ***xvmaddmdp***, ***xvmsubadp***, ***xvmsubmdp*** ***xvnmaddadp***, ***xvnmaddmdp***, ***xvnmsubadp***, ***xvnmsubmdp***
  VSX Vector Reciprocal Estimate Double-Precision (***xvredp***)
  do the following.
  1. UX is set to 1.
  2. For each vector element causing an Underflow exception, the result is placed into its respective doubleword element of VSR[XT] in double-precision format.
  3. FR, FI, and FPRF are not modified.
- For any of the following instructions,
  VSX Vector Single-Precision Arithmetic instructions
  xvaddsp, ***xvdivsp***, ***xvmulsp***, ***xvsubsp***, ***xvmaddasp***, ***xvmaddmsp***, ***xvmsubasp***, ***xvmsubmsp***, ***xvnmaddasp***, ***xvnmaddmsp***, ***xvnmsubasp***, ***xvnmsubmsp***
  VSX Vector Reciprocal Estimate Single-Precision (***xvresp***)
  VSX Vector round and Convert Double-Precision to Single-Precision format (***xvcvdpsp***)
  do the following.

1. UX is set to 1.
2. For each vector element causing an Underflow exception, the result is placed into its respective word element of VSR[XT] in single-precision format, and for ***xvcvdpsp***, is also placed into bits 32:63 of its respective doubleword element of VSR[XT].
3. FR, FI, and FPRF are not modified.

- For any of the following instructions,

  VSX Vector Convert with round Single-Precision to Half-Precision format (***xvcvsphp***)
  VSX Vector Convert with round Single-Precision to bfloat16 format (***xvcvspbf16***)

  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

Programming Note

In some implementations, enabling Inexact exceptions can degrade performance more than does enabling other types of floating-point exception.

When Inexact exception is enabled (XE=1) and an Inexact exception occurs, the following actions are taken:

- For the *VSX Scalar round and Convert Double-Precision to Single-Precision format* (***xscvdpsp***) instruction, do the following.
  1. XX is set to 1.
  2. The result is placed into word elements 0 and 1 of VSR[XT] in single-precision format. The contents of word elements 2-3 of VSR[XT] are set to 0.
  3. FPRF is set to indicate the class and sign of the result.
- For any of the following instructions,

  VSX Scalar Double-Precision Arithmetic instructions
  xsadddp, ***xssubdp***, ***xsmuldp***, ***xsdivdp***, ***xssqrtdp***, ***xsmaddadp***, ***xsmaddmdp***, ***xsmsubadp***, ***xsmsubmdp***, ***xsnmaddadp***, ***xsnmaddmdp***, ***xsnmsubadp***, ***xsnmsubmdp***
  VSX Scalar Single-Precision Arithmetic instructions
  xsaddsp, ***xssubsp***, ***xsmulsp***, ***xsdivsp***, ***xssqrtsp xsmaddasp***, ***xsmaddmsp***, ***xsmsubasp***, ***xsmsubmsp xsnmaddasp***, ***xsnmaddmsp***, ***xsnmsubasp***, ***xsnmsubmsp***
  VSX Scalar Reciprocal Estimate instructions
  xsredp, ***xsrsqrtedp***, ***xsresp***, ***xsrsqrtesp***
  VSX Scalar Round to Single-Precision (***xsrsp***)
  VSX Scalar Round to Double-Precision Integer Exact using Current rounding mode (***xsrdpic***)
  VSX Scalar Convert with round Signed Doubleword to Double-Precision format (***xscvsxddp***)
  VSX Scalar Convert with round Signed Doubleword to Single-Precision format (***xscvsxdsp***)
  VSX Scalar Convert with round Unsigned Doubleword to Double-Precision format (***xscvuxddp***)
  VSX Scalar Convert with round Unsigned Doubleword to Single-Precision format (***xscvuxdsp***)

  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 set to 0.
  3. FPRF is set to indicate the class and sign of the result.
- For any of the following instructions,

  VSX Scalar Convert with round to zero Double-Precision to Signed Word format (***xscvdpsxws***)
  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 set to 0.
  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***], ***xsdivqp***[***o***], ***xsmulqp***[***o***], ***xssqrtqp***[***o***], ***xssubqp***[***o***], ***xsmaddqp***[***o***], ***xsmsubqp***[***o***], ***xsnmaddqp***[***o***], ***xsnmsubqp***[***o***]
VSX Scalar Round to Quad-Precision Integer with Inexact (***xsrqpix***)
VSX Scalar Round Quad-Precision to Double-Extended Precision (***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* (***xscvqpudz***), 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* (***xscvqpuwz***), do the following.
  1. XX is set to 1.
  2. The result is placed into word element 1 of VSR[XT] in unsigned integer format.
     0x0000_0000 is placed into word elements 0, 2, and 3 of VSR[VRT+32].
  3. FR is set to 0. FI is set to 1. FPRF is undefined.
- For *VSX Scalar Convert with round Double-Precision to Half-Precision truncate* (***xscvdphp***), do the following.
  1. XX is set to 1.
  2. The result is placed into the rightmost halfword of doubleword element 0 of VSR[XT] as a half-precision value.
     The contents of the leftmost 3 halfwords of doubleword element 0 of VSR[XT] are set to 0.
     The contents of doubleword element 1 of VSR[XT] are set to 0.
  3. FR is set to indicate if the rounded result was incremented. FI is set to 1. FPRF is set to indicate the class and sign of the result.
- For any of the following instructions,
  VSX Scalar Double-Precision Arithmetic instructions
  xsadddp, ***xssubdp***, ***xsmuldp***, ***xsdivdp***, ***xssqrtdp*** ***xsmaddadp***, ***xsmaddmdp***, ***xsmsubadp***, ***xsmsubmdp*** ***xsnmaddadp***, ***xsnmaddmdp***, ***xsnmsubadp***, ***xsnmsubmdp***
  VSX Scalar Single-Precision Arithmetic instructions
  xsaddsp, ***xssubsp***, ***xsmulsp***, ***xsdivsp***, ***xssqrtsp*** ***xsmaddasp***, ***xsmaddmsp***, ***xsmsubasp***, ***xsmsubmsp*** ***xsnmaddasp***, ***xsnmaddmsp***, ***xsnmsubasp***, ***xsnmsubmsp***
  VSX Scalar Reciprocal Estimate instructions
  xsredp, ***xsrsqrtedp***, ***xsresp***, ***xsrsqrtesp***
  VSX Scalar Round to Single-Precision (***xsrsp***)
  VSX Scalar Round to Double-Precision Integer Exact using Current rounding mode (***xsrdpic***)

VSX Scalar Convert with round Signed Doubleword to Double-Precision format (***xscvsxddp***)
VSX Scalar Convert with round Unsigned Doubleword to Double-Precision format (***xscvuxddp***)
VSX Scalar Convert with round Signed Doubleword to Single-Precision format (***xscvsxdsp***)
VSX Scalar Convert with round Unsigned Doubleword to Single-Precision format (***xscvuxdsp***)

do the following.

1. XX 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 Convert with round Single-Precision to Half-Precision format (***xvcvsphp***)
  VSX Vector Convert with round Single-Precision to bfloat16 format (***xvcvspbf16***)

  do the following.

  1. XX is set to 1.
  2. VSR[XT] is not modified.
  3. FR, FI, and FPRF are not modified.

7.4.5.3 Action for XE=0

When Inexact exception is disabled (XE=0) and an Inexact exception occurs, the following actions are taken:

- For *VSX Scalar round and Convert Double-Precision to Single-Precision format* (***xscvdpsp***), do the following.
  1. XX is set to 1.
  2. The result is placed into word elements 0 and 1 of VSR[XT] as a single-precision value. The contents of word elements 2-3 of VSR[XT] are set to 0.
  3. FPRF is set to indicate the class and sign of the result.
- For any of the following instructions,

  VSX Scalar Double-Precision Arithmetic instructions
  xsadddp, ***xssubdp***, ***xsmuldp***, ***xsdivdp***, ***xssqrtdp*** ***xsmaddadp***, ***xsmaddmdp***, ***xsmsubadp***, ***xsmsubmdp*** ***xsnmaddadp***, ***xsnmaddmdp***, ***xsnmsubadp***, ***xsnmsubmdp***
  VSX Scalar Single-Precision Arithmetic instructions
  xsaddsp, ***xssubsp***, ***xsmulsp***, ***xsdivsp***, ***xssqrtsp*** ***xsmaddasp***, ***xsmaddmsp***, ***xsmsubasp***, ***xsmsubmsp*** ***xsnmaddasp***, ***xsnmaddmsp***, ***xsnmsubasp***, ***xsnmsubmsp***
  VSX Scalar Round to Single-Precision (***xsrsp***)
  VSX Scalar Round to Double-Precision Integer Exact using Current rounding mode (***xsrdpic***)
  VSX Scalar Convert with round Signed Doubleword to Double-Precision format (***xscvsxddp***)
  VSX Scalar Convert with round Unsigned Doubleword to Double-Precision format (***xscvuxddp***)

  do the following.

  1. XX is set to 1.
  2. The result is placed into doubleword element 0 of VSR[XT] as a double-precision value. The contents of doubleword element 1 of VSR[XT] are set to 0.
  3. FPRF is set to indicate the class and sign of the result.
- For any of the following instructions,

  VSX Scalar Convert with round to zero Double-Precision To Signed Word format (***xscvdpsxws***)
  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 elements 0 and 1 of VSR[XT]. The contents of word elements 2 and 3 of VSR[XT] are set to 0.
  3. FPRF is set to indicate the class and sign of the result.
- For *VSX Scalar Convert with round Quad-Precision to Double-Precision format* (***xscvqpdp***), do the following.
  1. XX is set to 1.
  2. The result is placed into the rightmost halfword of doubleword element 0 of VSR[XT] as a half-precision value.
     The contents of the leftmost 3 halfwords of doubleword element 0 of VSR[XT] are set to 0.
     The contents of doubleword element 1 of VSR[XT] are set to 0.
  3. FR is set to indicate if the rounded result was incremented. FI is set to 1. FPRF is set to indicate the class and sign of the result.

- For any of the following instructions,

  VSX Vector Double-Precision Arithmetic instructions
  xvadddp, ***xvsubdp***, ***xvmuldp***, ***xvdivdp***, ***xvsqrtdp***, ***xvmaddadp***, ***xvmaddmdp***, ***xvmsubadp***, ***xvmsubmdp***, ***xvnmaddadp***, ***xvnmaddmdp***, ***xvnmsubadp***, ***xvnmsubmdp***

  do the following.
  1. XX is set to 1.
  2. For each vector element causing an Inexact exception, the result is placed into its respective doubleword element of VSR[XT] in double-precision format.
  3. FR, FI, and FPRF are not modified.

- For any of the following instructions,

  VSX Scalar Quad-Precision Arithmetic instructions
  xsaddqp[***o***], ***xsdivqp***[***o***], ***xsmulqp***[***o***], ***xssqrtqp***[***o***], ***xssubqp***[***o***], ***xsmaddqp***[***o***], ***xsmsubqp***[***o***], ***xsnmaddqp***[***o***], ***xsnmsubqp***[***o***]
  VSX Scalar Round Quad-Precision to Double-Extended-Precision (***xsrqpxp***)
  VSX Scalar Round to Quad-Precision Integer with Inexact (***xsrqpix***)

  do the following.
  1. XX is set to 1.
  2. The result is placed into VSR[VRT+32] in quad-precision format.
  3. FR is set to indicate if the rounded result was incremented. FI is set to 1. FPRF is set to indicate the class and sign of the result.

- For *VSX Scalar round & Convert Quad-Precision to Double-Precision* (***xscvqpdp***), do the following.
  1. XX is set to 1.
  2. The result is placed into doubleword element 0 of VSR[VRT+32] in double-precision format.
     0x0000_0000_0000_0000 is placed into doubleword element 1 of VSR[VRT+32].
  3. FR is set to indicate if the rounded result was incremented. FI is set to 1. FPRF is set to indicate the class and sign of the result.

- For any of the following instructions,

  VSX Scalar truncate & Convert Quad-Precision to Signed Doubleword (***xscvqpsdz***)
  VSX Scalar truncate & Convert Quad-Precision to Signed Word (***xscvqpswz***)

  do the following.
  1. 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 (***xscvqpudz***)
  VSX Scalar truncate & Convert Quad-Precision to Unsigned Word (***xscvqpuwz***)

  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 any of the following instructions,

  VSX Vector Convert with round Single-Precision to Half-Precision format (***xvcvsphp***)
  VSX Vector Convert with round Single-Precision to bfloat16 format (***xvcvspbf16***)

  do the following.
  1. XX is set to 1.
  2. For each vector element causing an Inexact exception, the result is placed into the rightmost halfword of its respective word element of VSR[XT] in half-precision format.
     The contents of the leftmost halfword of its respective word element of VSR[XT] are set to 0.
  3. FR, FI, and FPRF are not modified.

- For any of the following instructions,

  VSX Vector Single-Precision Arithmetic instructions

xvaddsp, ***xvsubsp***, ***xvmulsp***, ***xvdivsp***, ***xvsqrtsp***, ***xvmaddasp***, ***xvmaddmsp***, ***xvmsubasp***, ***xvmsubmsp***, ***xvnmaddasp***, ***xvnmaddmsp***, ***xvnmsubasp***, ***xvnmsubmsp***

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.

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, AW, consists of 4 words.

```
int    AW[4] = { 0x00010203,
                 0x04050607,
                 0x08090A0B,
                 0x0C0D0E0F };
```

Figure 7.17 illustrates the Big-Endian storage image of array AW.

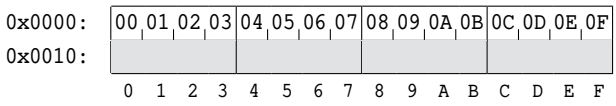


Figure 7.17: Big-Endian storage image of array AW

Figure 7.18 illustrates the Little-Endian storage image of array AW.

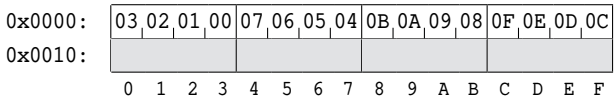


Figure 7.18: Little-Endian storage image of array AW

Figure 7.19 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 7.17 for Big-Endian. Note that Figure 7.19 shows the effect of loading the quadword from both Big-Endian storage and Little-Endian storage.

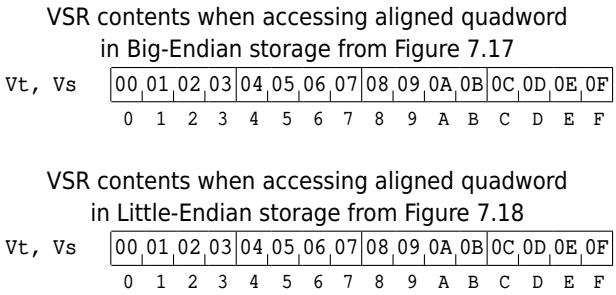


Figure 7.19: Vector-Scalar Register contents for aligned quadword Load or Store VSX Vector

7.5.2 Accessing Unaligned Storage Operands

The following array, B, consists of 5 word elements.

```
int   B[5];
B[0] = 0x01234567;
B[1] = 0x00112233;
B[2] = 0x44556677;
B[3] = 0x8899AABB;
B[4] = 0xCCDDEEFF;
```

Figure 7.20 illustrates both Big-Endian and Little-Endian storage images of array B.

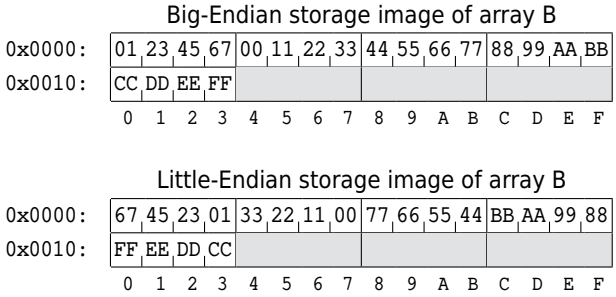


Figure 7.20: Storage images of array B

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 7.20) into VR[VT] involves an unaligned quadword storage access.

VSX supports word-aligned vector and scalar storage accesses using Big-Endian byte ordering.

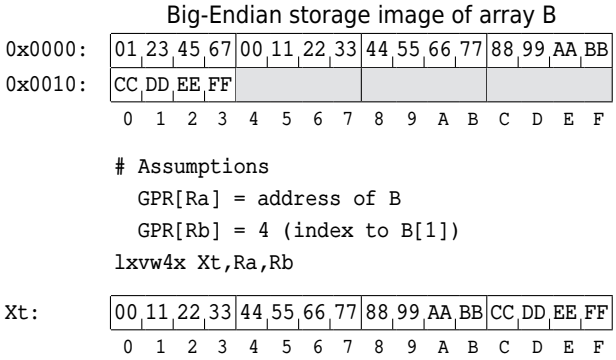


Figure 7.21: Process to load unaligned quadword from Big-Endian storage using Load VSX Vector Word*4 Indexed

Loading an Unaligned Quadword from Little-Endian Storage

Loading elements from elements 1 through 4 of B (see Figure 7.20) into VR[VT] involves an unaligned quadword storage access.

VSX supports word-aligned vector and scalar storage accesses using Little-Endian byte ordering.

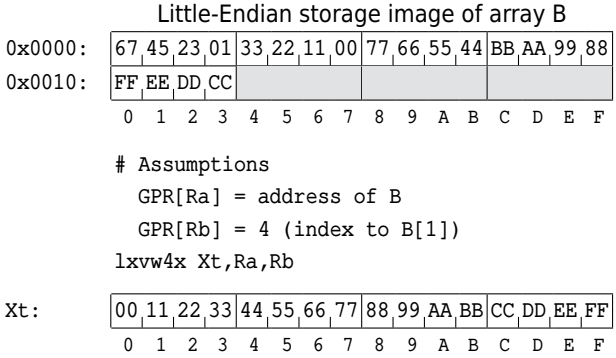


Figure 7.22: Process to load unaligned quadword from Little-Endian storage Load VSX Vector Word*4 Indexed

Storing an Unaligned Quadword to Big-Endian Storage

Storing a VSR to elements 1 through 4 of B (see Figure 7.20) into VR[VT] involves an unaligned quadword storage access.

VSX supports word-aligned vector and scalar storage accesses using Big-Endian byte ordering.

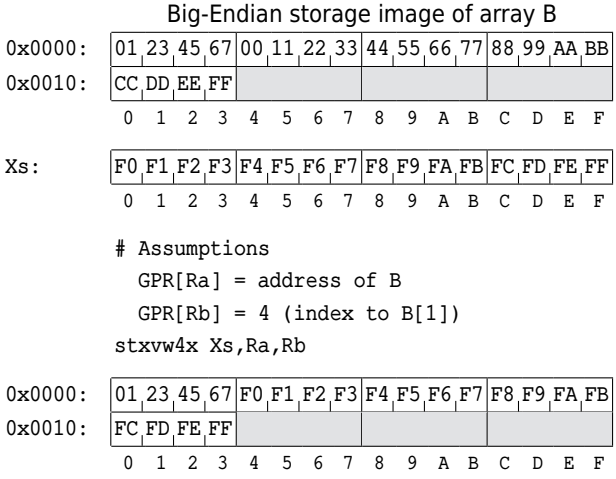


Figure 7.23: Process to store unaligned quadword to Big-Endian storage using Store VSX Vector Word*4 Indexed

Storing an Unaligned Quadword to Little-Endian Storage

Storing a VSR to elements 1 through 4 of B (see Figure 7.20) into VR[VT] involves an unaligned quadword storage access.

VSX supports word-aligned vector and scalar storage accesses using Little-Endian byte ordering.

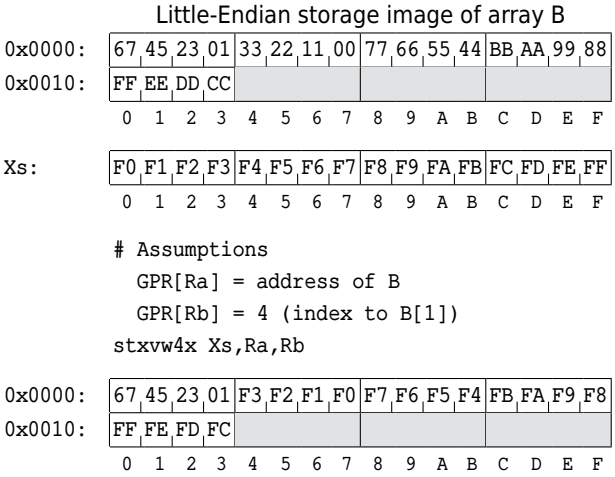


Figure 7.24: Process to store unaligned quadword to Little-Endian storage Store VSX Vector Word*4 Indexed

7.5.3 Storage Access Exceptions

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

7.6 VSX Instruction Set

7.6.1 VSX Instruction Description Conventions

7.6.1.1 VSX Instruction RTL Operators

x.bit[y:z]
Return the contents of bits `y:z` of `x`.

x.word[y:z]
Return the contents of word elements `y:z` of `x`.

x.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.

¬x Return the one's complement of x.

!x Return 1 if the contents of x are equal to 0, otherwise return 0.

x || y
Return the value of `x` concatenated with the value of `y`. For example, `0b010 || 0b111` is the same as `0b010111`.

x ⊕ y
Return the value of x exclusive ORed with the value of y.

x ? y : z
If the value of x is true, return the value of y, otherwise return the value z.

x + y
x and y are integer values.
Return the sum of x and y.

x - y x and y are integer values.
Return the difference of x and y.

x != y
x and y are integer values.
Return 1 if x is not equal to y, otherwise return 0.

x ≤ y
x and y are integer values.
Return 1 if x is less than or equal to y, otherwise return 0.

x ≥ y
x and y are integer values.
Return 1 if x is greater than or equal to y, otherwise return 0.

7.6.1.2 VSX Instruction RTL Function Calls

`bfloat16_CONVERT_FROM_BFP(x)`
`x` is a floating-point value represented in the working format.
If `x.class.SNaN=1` or `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 `0b11111111`.
Bits 9:15 of `result` are set to the value of bits 1:7 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 1:8 of `result` are set to the value `0b11111111`.
Bits 9:15 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 9:15 of `result` are set to 0.
Otherwise, if `x.exponent` is less than `-126`, do the following.
Bit 0 of `result` is set to the value of `x.sign`. `sh_cnt` is set to the difference, `-126 − x.exponent`.
Bits 1:8 of result are set to `0b00000000`.
Bits 9:15 of result are set to bits 1:7 of `x.significand` shifted right by `sh_cnt` bits.
Otherwise, do the following.
Bit 0 of `result` is set to the value of `x.sign`.
Bits 1:8 of `result` are set to the sum, `x.exponent + 127`.
Bits 9:15 of `result` are set to bits 1:7 of `x.significand`.
Return `result` (bfloat16 format)

`bfp_ABSOLUTE(x)`
`x` is a binary floating-point value represented in the binary floating-point working format.
Return `x` with sign set to 0.

bfp_ADD(x, y)
x is a binary floating-point value represented in the binary floating-point working format.
y is a binary floating-point value represented in the binary floating-point working format.
If x or y is an SNaN, vxsnan_flag is set to 1.
If x is an infinity and y is an infinity of the opposite sign, vxisi_flag is set to 1.
If x is a QNaN, return x.
Otherwise, if x is an SNaN, return x represented as a QNaN.
Otherwise, if y is a QNaN, return y.
Otherwise, if y is an SNaN, return y represented as a QNaN.
Otherwise, if x and y are infinities of opposite sign, return the standard QNaN.
Otherwise, return the normalized sum of x and y, having unbounded range and precision.

bfp_COMPARE_EQ(x, y)
x is a binary floating-point value represented in the binary floating-point working format.
y is a binary floating-point value represented in the binary floating-point working format.
Return 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 binary floating-point working format.
y is a binary floating-point value represented in the binary floating-point working format.
Return 0b0 if x is NaN or y is a NaN.
Otherwise, return 0b0 if x is a Zero and y is a Zero.
Otherwise, return 0b1 if x is greater than y.
Otherwise, return 0b0.

bfp_COMPARE_LT(x, y)
x is a binary floating-point value represented in the binary floating-point working format.
y is a binary floating-point value represented in the binary floating-point working format.
Return 0b0 if x is NaN or y is a NaN.
Otherwise, return 0b0 if x is a Zero and y is a Zero.
Otherwise, return 0b1 if x is less than y.
Otherwise, return 0b0.

bfp_CONVERT_FROM_BFLOAT16(x)
x is a floating-point value represented in bfloat16 format.
Let sign be the contents of bit 0 of x.
Let exponent be the contents of bits 1:8 of x.
Let fraction be the contents of bits 9:15 of x.
Let result.sign be set to 0.
Let result.exponent be set to 0.
Let result.significand be set to 0.
Let result.class.SNaN be set to 0.
Let result.class.QNaN be set to 0.
Let result.class.Infinity be set to 0.
Let result.class.Zero be set to 0.
Let result.class.Denormal be set to 0.
Let result.class.Normal be set to 0.
If x is an SNaN, do the following.
result.class.SNaN is set to 1.
result.sign is set to the value of sign.
The contents of result.significand are set to 0.
The contents of bits 1:7 of result.significand are set to the value of fraction.
Otherwise, if x is a QNaN, do the following.
result.class.QNaN is set to 1.
result.sign is set to the value of sign.

The contents of result.significand are set to 0.
The contents of bits 1:7 of result.significand are set to the value of fraction.

Otherwise, if x is an Infinity value, do the following.
result.class.Infinity is set to 1.
result.sign is set to the value of sign.

Otherwise, if x is a Zero value, do the following.
result.class.Zero is set to 1.
result.sign is set to the value of sign.

Otherwise, if x is a Denormal value, do the following.
result.class.Denormal is set to 1.
result.sign is set to the value of sign.
result.exponent is set to the value –126.
The contents of bits 1:7 of result.significand are set to the value of fraction.
result.significand is shifted left until the contents bit 0 of result.significand are equal to 1.
result.exponent is decremented by the number of bits result.significand was shifted.

Otherwise, do the following.
result.class.Normal is set to 1.
result.sign is set to the value of sign.
result.exponent is set to the value of exponent subtracted by 127.
The contents of bit 0 of result.significand are set to 1.
The contents of bits 1:7 of result.significand are set to the value of fraction.

Return result (binary floating-point working format).

bfp_CONVERT_FROM_BFP16(x)

x is a floating-point value represented in half-precision format.

Let exponent be the contents of bits 1:5 of x.
Let fraction be the contents of bits 6:15 of x.

Let result.sign be set to 0.
Let result.exponent be set to 0.
Let result.significand be set to 0.
Let result.class.SNaN be set to 0.
Let result.class.QNaN be set to 0.
Let result.class.Infinity be set to 0.
Let result.class.Zero be set to 0.
Let result.class.Denormal be set to 0.
Let result.class.Normal be set to 0.

If x is a SNaN, do the following.
result.class.SNaN is set to 1.
result.sign is set to the contents of bit 0 of x.
The contents of bit 0 of result.significand are set to 0.
The contents of bits 1:10 of result.significand are set to the value of fraction.

Otherwise, if x is a QNaN, do the following.
result.class.QNaN is set to 1.
result.sign is set to the contents of bit 0 of x.
The contents of bit 0 of result.significand are set to 0.
The contents of bits 1:10 of result.significand are set to the value of fraction.

Otherwise, if x is an Infinity value, do the following.
result.class.Infinity is set to 1.
result.sign is set to the contents of bit 0 of x.

Otherwise, if x is a Zero value, do the following.
result.class.Zero is set to 1.
result.sign is set to the contents of bit 0 of x.

Otherwise, if x is a Denormal value, do the following.
result.class.Denormal is set to 1.
result.sign is set to the contents of bit 0 of x.
result.exp is set to the value -14.
The contents of bit 0 of result.significand are set to 0.
The contents of bits 1:10 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.exp is set to the value of exponent subtracted by 15.
The contents of bit 0 of result.significand are set to 1.
The contents of bits 1:10 of result.significand are set to the value of fraction.

Return 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 binary floating-point working format).

bfp_CONVERT_FROM_BFP128(x)

x is a binary floating-point value represented in quad-precision format.

Let exponent be the contents of bits 1:15 of x.
Let fraction be the contents of bits 16:127 of x.

result.sign is initialized to 0.
result.exponent is initialized to 0.
result.significand is initialized to 0.
result.class.SNaN is initialized to 0.
result.class.QNaN is initialized to 0.
result.class.Infinity is initialized to 0.
result.class.Zero is initialized to 0.
result.class.Denormal is initialized to 0.
result.class.Normal is initialized to 0.

If x is a SNaN, do the following.

result.class.SNaN is set to 1.
result.sign is set to the contents of bit 0 of x.
The contents of bit 0 of result.significand are set to 0.
The contents of bits 1:112 of result.significand are set to the value of fraction.
The contents of the rest of result.significand are set to 0.

Otherwise, if x is a QNaN, do the following.
result.class.QNaN is set to 1.
result.sign is set to the contents of bit 0 of x.
The contents of bit 0 of result.significand are set to 0.
The contents of bits 1:112 of result.significand are set to the value of fraction.
The contents of the rest of result.significand are set to 0.

Otherwise, if x is an Infinity, do the following.
result.class.Infinity is set to 1.
result.sign is set to the contents of bit 0 of x.

Otherwise, if x is a Zero, do the following.
result.class.Zero is set to 1.
result.sign is set to the contents of bit 0 of x.

Otherwise, if x is a Denormal, do the following.
result.class.Denormal is set to 1.
result.sign is set to the contents of bit 0 of x.
result.exp is set to the value -16382.
The contents of bit 0 of result.significand are set to 0.
The contents of bits 1:112 of result.significand are set to the value of fraction.
The contents of the rest of result.significand are set to 0.
result.significand is shifted left until the contents bit 0 of result.significand are equal to 1.
result.exponent is decremented by the the number of bits result.significand was shifted.

Otherwise, do the following.
result.class.Normal is set to 1.
result.sign is set to the contents of bit 0 of x.
result.exp is set to the value of exponent subtracted by 16383.
The contents of bit 0 of result.significand are set to 1.
The contents of bits 1:112 of result.significand are set to the value of fraction.
The contents of the rest of result.significand are set to 0.

Return result (i.e., the value x in the binary floating-point working format).

bfp_CONVERT_FROM_SI64(x)

x is an integer value represented in signed doubleword integer format.
result.sign is initialized to 0.
result.exponent is initialized to 0.
result.significand is initialized to 0.
result.class.SNaN is initialized to 0.
result.class.QNaN is initialized to 0.
result.class.Infinity is initialized to 0.
result.class.Zero is initialized to 0.
result.class.Denormal is initialized to 0.
result.class.Normal is initialized to 0.

If x is equal to 0x0000_0000_0000_0000,
result.class.Zero is set to 1.

Otherwise, do the following.
result.class.Normal is set to 1.
result.sign is set to the contents of bit 0 of x.
result.exponent is set to the value 64.
Bits 0:64 of result.significand are set to the value of x sign-extended to 65 bits.

If bit 0 of result.significand is equal to 1,
result.sign is set to 1, and
result.significand is set to the value of the two's complement of result.significand.

If bit 0 of result.significand is equal to 0,

result.significand is shifted left until bit 0 of result.significand is equal to 1, and
result.exponent is decremented by the number of bits result.significand is shifted.

Return result (i.e., the value x in the binary floating-point working format).

bfp_CONVERT_FROM_SI128(x)

x is a 128-bit signed integer value.

result.sign is initialized to 0.
result.exponent is initialized to 0.
result.significand is initialized to 0.
result.class.SNaN is initialized to 0.
result.class.QNaN is initialized to 0.
result.class.Infinity is initialized to 0.
result.class.Zero is initialized to 0.
result.class.Denormal is initialized to 0.
result.class.Normal is initialized to 0.

If x is equal to 0x0000_0000_0000_0000_0000_0000_0000_0000,
result.class.Zero is set to 1.

Otherwise, do the following.
result.class.Normal is set to 1.
result.sign is set to the contents of bit 0 of x.
result.exponent is set to the value 128.
Bits 0:128 of result.significand are set to the value of x sign-extended to 129 bits.

If bit 0 of result.significand is equal to 1,
result.sign is set to 1, and
result.significand is set to the value of the two's complement of result.significand.

If bit 0 of result.significand is equal to 0,
result.significand is shifted left until bit 0 of result.significand is equal to 1, and
result.exponent is decremented by the number of bits result.significand is shifted.

Return result (i.e., the value x in the binary floating-point working format).

bfp_CONVERT_FROM_UI64(x)

x is an integer value represented in unsigned doubleword integer format.

result.sign is initialized to 0.
result.exponent is initialized to 0.
result.significand is initialized to 0.
result.class.SNaN is initialized to 0.
result.class.QNaN is initialized to 0.
result.class.Infinity is initialized to 0.
result.class.Zero is initialized to 0.
result.class.Denormal is initialized to 0.
result.class.Normal is initialized to 0.

If x is equal to 0x0000_0000_0000_0000, do the following.
result.class.Zero is set to 1.

Otherwise, do the following.
result.class.Normal is set to 1.
result.sign is set to 0.
result.exponent is set to the value 64.
Bits 0:64 of result.significand is set to the value of x zero-extended to 65 bits.

If bit 0 of result.significand is equal to 0, result.significand is shifted left until bit 0 of result.significand is equal to 1 and result.exponent is decremented by the number of bits result.significand is shifted.

Return result (i.e., the value x in the binary floating-point working format).

bfp_CONVERT_FROM_UI128(x)

x is a 128-bit unsigned integer value.

result.sign is initialized to 0.
result.exponent is initialized to 0.

result.significand is initialized to 0.
result.class.SNaN is initialized to 0.
result.class.QNaN is initialized to 0.
result.class.Infinity is initialized to 0.
result.class.Zero is initialized to 0.
result.class.Denormal is initialized to 0.
result.class.Normal is initialized to 0.
If x is equal to 0x0000_0000_0000_0000_0000_0000_0000_0000, do the following.
result.class.Zero is set to 1.
Otherwise, do the following.
result.class.Normal is set to 1.
result.sign is set to 0.
result.exponent is set to the value 128.
Bits 0:128 of result.significand are set to the value of x zero-extended to 129 bits.

If bit 0 of result.significand is equal to 0,
result.significand is shifted left until bit 0 of result.significand is equal to 1 and result.exponent is decremented by the number of bits result.significand is shifted.
Return result (i.e., the value x in the binary floating-point working format).

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 binary floating-point working 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 binary floating-point working format.

bfp_DIVIDE(x, y)
x is a binary floating-point value that is represented in the binary floating-point working format.
y is a binary floating-point value that is represented in the binary floating-point working format.
If x or y is an SNaN, 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
The value +Infinity represented in the binary floating-point working format.

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 binary floating-point working format.
y is a binary floating-point value represented in the binary floating-point working format.

If x or y is an SNaN, vxsnan_flag is set to 1.
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 $x \times y$, having unbounded range and precision.

bfp_MULTIPLY_ADD(x, y, z)

x is a binary floating-point value represented in the binary floating-point working format.
y is a binary floating-point value represented in the binary floating-point working format.
z is a binary floating-point value represented in the binary floating-point working format.

If x, y, or z is an SNaN, 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.
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.

bfp_NEGATE(x)

x is a binary floating-point value that is represented in the binary floating-point working format.

If x is not a NaN, return x with its sign complemented. Otherwise, return x.

bfp_NMAX_BFLOAT16

Return the largest positive normalized bfloat16 floating-point value (i.e., $2^{128} - 2^{128-8}$) represented in the binary floating-point working format.

```
return bfp_CONVERT_FROM_BFLOAT16(0x7F7F)
```

bfp_NMAX_BFP16

Return the largest positive normalized half-precision floating-point value (i.e., $2^{16} - 2^{16-11}$), represented in the binary floating-point working format.

```
return bfp_CONVERT_FROM_BFP16(0x7BFF)
```

bfp_NMAX_BFP64

Return the largest finite double-precision floating-point value (i.e., $2^{1024} - 2^{1024-53}$) in the binary floating-point working format.

```
return bfp_CONVERT_FROM_BFP64(0x7FEF_FFFF_FFFF_FFFF)
```

bfp_NMAX_BFP80

Return the largest finite double-extended-precision floating-point value (i.e., $2^{16384} - 2^{16384-65}$) in the binary floating-point working format.

```
return bfp_CONVERT_FROM_BFP80(0x7FFE_FFFF_FFFF_FFFF_FFFF)
```

bfp_NMAX_BFP128
: Return the largest finite quad-precision value (i.e., $2^{16384} - 2^{16384-113}$) in the binary floating-point working format.

```
return bfp_CONVERT_FROM_BFP128(0x7FFE_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF)
```

bfp_NMIN_BFLOAT16
: Return the smallest positive normalized bfloat16 floating-point value (i.e., 2^{-126}), represented in the binary floating-point working format.

```
return bfp_CONVERT_FROM_BFLOAT16(0x0080)
```

bfp_NMIN_BFP16
: Return the smallest positive normalized half-precision floating-point value, 2^{-14}, represented in the binary floating-point working format.

```
return bfp_CONVERT_FROM_BFP16(0x0400)
```

bfp_NMIN_BFP64
: Return the smallest positive normalized double-precision floating-point value, 2^{-1022}, represented in the binary floating-point working format.

```
return bfp_CONVERT_FROM_BFP64(0x0010_0000_0000_0000)
```

bfp_NMIN_BFP80
: Return the smallest positive normalized double-extended-precision floating-point value, 2^{-16382}, represented in the binary floating-point working format.

```
return bfp_CONVERT_FROM_BFP80(0x0001_0000_0000_0000_0000)
```

bfp_NMIN_BFP128
: Return the smallest, positive, normalized quad-precision floating-point value, 2^{-16382}, represented in the binary floating-point working format.

```
return bfp_CONVERT_FROM_BFP128(0x0001_0000_0000_0000_0000_0000_0000_0000)
```

bfp_QUIET(x)
: x is a Signalling NaN.

  Return x converted to a Quiet NaN with x.class.QNaN set to 1 and x.class.SNaN set to 0.

bfp_ROUND_CEIL(p, x)
: x is a binary floating-point value that is represented in the binary floating-point working format and has unbounded exponent range and significand precision. x must be rounded as presented, without prenormalization.

  p is an integer value specifying the precision (i.e., number of bits) the significand is rounded to.

  Return the smallest floating-point number having unbounded exponent range and a significand with a width of p bits that is greater or equal in value to x.

  inc_flag is set to 1 if the magnitude of the value returned is greater than x.
  xx_flag is set to 1 if the value returned is not equal to x.

bfp_ROUND_FLOOR(p, x)
: x is a binary floating-point value that is represented in the binary floating-point working format and has unbounded exponent range and significand precision. The value must be rounded as presented, without prenormalization.

  p is an integer value specifying the precision (i.e., number of bits) the significand is rounded to.

  Return the largest floating-point number having unbounded exponent range and a significand with a width of p bits that is lesser or equal in value to x.

inc_flag is set to 1 if the magnitude of the value returned is greater than x.
xx_flag is set to 1 if the value returned is not equal to x.

bfp_ROUND_TO_BFLOAT16(rmode, x)

x is a normalized binary floating-point value that is represented in the binary floating-point working format and has unbounded exponent range and significand precision.

rmode is a 2-bit integer value specifying one of four rounding modes.

rmode=0b00 Round to Nearest Even
rmode=0b01 Round towards Zero
rmode=0b10 Round towards + Infinity
rmode=0b11 Round towards – Infinity

If x is a QNaN, Infinity, or Zero, return x.
Otherwise, if x is an SNaN, set vxsnan_flag to 1 and return the corresponding QNaN representation of x.
Otherwise, return the value x rounded to bfloat16 format's exponent range and significand precision represented in the floating-point working format using the rounding mode specified by rmode.

```
if x.class.SNaN then do
   vxsnan_flag ← 1
   return bfp_QUIET(x)
end
if x.class.QNaN then return x
if x.class.Infinity then return x
if x.class.Zero then return x
if bfp_ABSOLUTE(x) < bfp_NMIN_BFLOAT16 then do
   if FPSCR.UE=0 then do
      x ← bfp_DENORM(-126,x)
      if rmode=0b00 then r ← bfp_ROUND_NEAR_EVEN(8,x)
      if rmode=0b01 then r ← bfp_ROUND_TRUNC(8,x)
      if rmode=0b10 then r ← bfp_ROUND_CEIL(8,x)
      if rmode=0b11 then r ← bfp_ROUND_FLOOR(8,x)
      ux_flag ← xx_flag
      return r
   end
   else do
      x.exponent ← x.exponent + 192
      ux_flag ← 0b1
   end
end
if rmode=0b00 then r ← bfp_ROUND_NEAR_EVEN(8,x)
if rmode=0b01 then r ← bfp_ROUND_TRUNC(8,x)
if rmode=0b10 then r ← bfp_ROUND_CEIL(8,x)
if rmode=0b11 then r ← bfp_ROUND_FLOOR(8,x)
if bfp_ABSOLUTE(r) > bfp_NMAX_BFLOAT16 then do
   if 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_BFLOAT16 : bfp_NMAX_BFLOAT16
      if rmode=0b10 then r ← x.sign ? bfp_NMAX_BFLOAT16 : bfp_INFINITY
      if rmode=0b11 then r ← x.sign ? bfp_INFINITY : bfp_NMAX_BFLOAT16
      r.sign ← x.sign
      ox_flag ← 0b1
      xx_flag ← 0b1
      inc_flag ← 0bU
   end
   else do
      x.exponent ← x.exponent - 192
      ox_flag ←  0b1
   end
end
return r
```

bfp_ROUND_TO_BFP16(rmode, x)

x is a normalized floating-point value represented in the binary floating-point working format, having unbounded exponent range and significand precision.

rmode is a 2-bit integer value specifying one of four rounding modes.

rmode=0b00	Round to Nearest Even
rmode=0b01	Round towards Zero
rmode=0b10	Round towards + Infinity
rmode=0b11	Round towards – Infinity

If x is an QNaN, Infinity, or Zero, return x.
Otherwise, if x is an SNaN, set vxsnan_flag to 1 and return the corresponding QNaN representation of x.
Otherwise, return the value x rounded to half-precision format's exponent range and significand precision using the rounding mode specified by rmode.

```
if x.class.SNaN then do
   vxsnan_flag ← 1
   return bfp_QUIET(x)
end
if x.class.QNaN then return x
if x.class.Infinity then return x
if x.class.Zero then return x
if bfp_ABSOLUTE(x) < bfp_NMIN_BFP16) then do
   if FPSCR.UE=0 then do
      do while x.exponent < -14    // denormalize x
         x.significand ← x.significand >> 1
         x.exponent ← x.exponent + 1
      end
      if rmode=0b00 then result ← bfp_ROUND_TO_BFP16_NEAR_EVEN(x)
      if rmode=0b01 then result ← bfp_ROUND_TO_BFP16_TRUNC(x)
      if rmode=0b10 then result ← bfp_ROUND_TO_BFP16_CEIL(x)
      if rmode=0b11 then result ← bfp_ROUND_TO_BFP16_FLOOR(x)
      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
      x.exponent ← x.exponent + 24
      ux_flag ← 1
   end
end
if rmode=0b00 then result ← bfp_ROUND_TO_BFP16_NEAR_EVEN(x)
if rmode=0b01 then result ← bfp_ROUND_TO_BFP16_TRUNC(x)
if rmode=0b10 then result ← bfp_ROUND_TO_BFP16_CEIL(x)
if rmode=0b11 then result ← bfp_ROUND_TO_BFP16_FLOOR(x)
if bfp_ABSOLUTE(result) > bfp_NMAX_BFP16 then do
   if OE=0 then do
      if rmode=0b00 then result ← sign ? bfp_NEGATE(bfp_INFINITY)   : bfp_INFINITY
      if rmode=0b01 then result ← sign ? bfp_NEGATE(bfp_NMAX_BFP16) : bfp_NMAX_BFP16
      if rmode=0b10 then result ← sign ? bfp_NEGATE(bfp_NMAX_BFP16) : bfp_INFINITY
      if rmode=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 binary floating-point working format, having unbounded exponent range and significand precision.

Return the smallest floating-point number having unbounded exponent range but half-precision significand precision that is greater or equal in value to x.

If the magnitude of the value returned is greater than x, inc_flag is set to 1.

If the value returned is not equal to x, xx_flag is set to 1.

bfp_ROUND_TO_BFP16_FLOOR(x)

x is a normalized floating-point value represented in the binary floating-point working format, having unbounded exponent range and significand precision.

Return the largest floating-point number having unbounded exponent range but half-precision significand precision that is lesser or equal in value to x.

If the magnitude of the value returned is greater than x, inc_flag is set to 1.

If the value returned is not equal to x, xx_flag is set to 1.

bfp_ROUND_TO_BFP16_NEAR_EVEN(x)

x is a normalized floating-point value represented in the binary floating-point working format, having unbounded exponent range and significand precision.

Return the floating-point number having unbounded exponent range but half-precision significand precision that is nearest in value to x (in case of a tie, the floating-point number having unbounded exponent range but half-precision significand precision with the least-significant bit equal to 0 is used).

If the magnitude of the value returned is greater than x, inc_flag is set to 1.

If the value returned is not equal to x, xx_flag is set to 1.

bfp_ROUND_TO_BFP16_TRUNC(x)

x is a normalized floating-point value represented in the binary floating-point working format, having unbounded exponent range and significand precision.

Return the largest floating-point number having unbounded exponent range but half-precision significand precision that is lesser or equal in value to x if x>0, or the smallest floating-point number having unbounded exponent range but half0-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_BFP32_SIGNIFICAND(x)

x is a normalized binary floating-point value that is represented in the binary floating-point working format and has unbounded exponent range and significand precision.

Return the value x rounded to 24-bit significand precision under control of the rounding mode specified in RN, retaining unbounded exponent range, represented in the binary floating-point working format.

```
rmode ← FPSCR.RN
if x.class.QNaN | x.class.Infinity | x.class.Zero then return x
if rmode=0b00 then r ← bfp_ROUND_NEAR_EVEN(24,x)
if rmode=0b01 then r ← bfp_ROUND_TRUNC(24,x)
if rmode=0b10 then r ← bfp_ROUND_CEIL(24,x)
if rmode=0b11 then r ← bfp_ROUND_FLOOR(24,x)
return r  (binary floating-point working format)
```

bfp_ROUND_TO_BFP32(rmode, x)

x is a normalized binary floating-point value that is represented in the binary floating-point working format and has unbounded exponent range and significand precision.

rmode is a 2-bit integer value specifying one of four rounding modes.

rmode=0b00 Round to Nearest Even

rmode=0b01 Round towards Zero
rmode=0b10 Round towards + Infinity
rmode=0b11 Round towards – Infinity

If x is a QNaN, Infinity, or Zero, return x. Otherwise, if x is an SNaN, set vxsnan_flag to 1 and return the corresponding QNaN representation of x. Otherwise, return the value x rounded to single-precision format's exponent range and significand precision represented in the floating-point working format using the rounding mode specified by rmode.

```
if x.class.SNaN then do
   vxsnan_flag ← 1
   return bfp_QUIET(x)
end
if x.class.QNaN then return x
if x.class.Infinity then return x
if x.class.Zero then return x
if bfp_ABSOLUTE(x) < bfp_NMIN_BFP32 then do
   x = bfp_DENORM(-126,x)
   if rmode=0b00 then r = bfp_ROUND_NEAR_EVEN(24,x)
   if rmode=0b01 then r = bfp_ROUND_TRUNC(24,x)
   if rmode=0b10 then r = bfp_ROUND_CEIL(24,x)
   if rmode=0b11 then r = bfp_ROUND_FLOOR(24,x)
   if FPSCR.UE=0 then do
      ux_flag ← xx_flag
      return(r)
   end
   else do
      x.exponent ← x.exponent + 192
      ux_flag ← 0b1
   end
end
if rmode=0b00 then r = bfp_ROUND_NEAR_EVEN(24,x)
if rmode=0b01 then r = bfp_ROUND_TRUNC(24,x)
if rmode=0b10 then r = bfp_ROUND_CEIL(24,x)
if rmode=0b11 then r = bfp_ROUND_FLOOR(24,x)

if bfp_ABSOLUTE(r) > bfp_NMAX_BFP32 then do
   if 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_BFP32 : bfp_NMAX_BFP32
      if rmode=0b10 then r ← x.sign ? bfp_NMAX_BFP32 : bfp_INFINITY
      if rmode=0b11 then r ← x.sign ? bfp_INFINITY   : bfp_NMAX_BFP64
      r.sign   ← x.sign
      ox_flag  ← 0b1
      xx_flag  ← 0b1
      inc_flag ← 0bU
      return(r)
   end
   else do
      r.exponent ← r.exponent - 192
      ox_flag ← 0b1
   end
end
return r
```

bfp_ROUND_TO_BFP32_DEFAULT(rmode, x)

x is a normalized binary floating-point value that is represented in the binary floating-point working format and has unbounded exponent range and significand precision.

rmode is a 2-bit integer value specifying one of four rounding modes.

rmode=0b00 Round to Nearest Even
rmode=0b01 Round towards Zero
rmode=0b10 Round towards + Infinity

rmode=0b11 Round towards – Infinity

If x is a QNaN, Infinity, or Zero, return x.
Otherwise, if x is an SNaN, set vxsnan_flag to 1 and return the corresponding QNaN representation of x.
Otherwise, return the value x rounded to single-precision format's exponent range and significand precision represented in the floating-point working format using the rounding mode specified by rmode.
The result is that which would be produced if OE and UE were set to 0 (the "default" value).

```
if x.class.SNaN then do
   vxsnan_flag ← 1
   return bfp_QUIET(x)
end
if x.class.QNaN then return x
if x.class.Infinity then return x
if x.class.Zero then return x
if bfp_ABSOLUTE(x) < bfp_NMIN_BFP32 then do
   x = bfp_DENORM(-126,x)
   if rmode=0b00 then r = bfp_ROUND_NEAR_EVEN(24,x)
   if rmode=0b01 then r = bfp_ROUND_TRUNC(24,x)
   if rmode=0b10 then r = bfp_ROUND_CEIL(24,x)
   if rmode=0b11 then r = bfp_ROUND_FLOOR(24,x)
   if FPSCR.UE=0 then do
      ux_flag ← xx_flag
   else do
      ux_flag ←  0b1

   return r
end
if rmode=0b00 then r = bfp_ROUND_NEAR_EVEN(24,x)
if rmode=0b01 then r = bfp_ROUND_TRUNC(24,x)
if rmode=0b10 then r = bfp_ROUND_CEIL(24,x)
if rmode=0b11 then r = bfp_ROUND_FLOOR(24,x)
if bfp_ABSOLUTE(r) > bfp_NMAX_BFP32 then do
   if rmode=0b00 then r = x.sign ? bfp_INFINITY   : bfp_INFINITY
   if rmode=0b01 then r = x.sign ? bfp_NMAX_BFP32 : bfp_NMAX_BFP32
   if rmode=0b10 then r = x.sign ? bfp_NMAX_BFP32 : bfp_INFINITY
   if rmode=0b11 then r = x.sign ? bfp_INFINITY   : bfp_NMAX_BFP32
   r.sign   = x.sign
   if FPSCR.OE=0 then do
      ox_flag ← 0b1
      xx_flag ← 0b1
      inc_flag ← 0bU
   else do
      ox_flag ←  0b1
end
return r
```

bfp_ROUND_TO_BFP64(ro,rmode,x)

x is a normalized binary floating-point value that is represented in the binary floating-point working format and has unbounded exponent range and significand precision.

ro is a 1-bit unsigned integer and rmode is a 2-bit unsigned integer, together specifying one of five rounding modes to be used in rounding z.

ro=0 rmode=0b00 Round to Nearest Even
ro=0 rmode=0b01 Round towards Zero
ro=0 rmode=0b10 Round towards + Infinity
ro=0 rmode=0b11 Round towards – Infinity
ro=1 Round to Odd

Return the value x rounded to double-precision under control of the specified rounding mode.

```
if x.class.SNaN then do
   vxsnan_flag ← 1
```

```
      return bfp_QUIET(x)
   end

   if x.class.QNaN     then return x
   if x.class.Infinity then return x
   if x.class.Zero     then return x
   if bfp_ABSOLUTE(x) < bfp_NMIN_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 ro=0 & rmode=0b00 then r ← bfp_ROUND_NEAR_EVEN(53,x)
   if ro=0 & rmode=0b01 then r ← bfp_ROUND_TRUNC(53,x)
   if ro=0 & rmode=0b10 then r ← bfp_ROUND_CEIL(53,x)
   if ro=0 & rmode=0b11 then r ← bfp_ROUND_FLOOR(53,x)
   if ro=1              then r ← bfp_ROUND_ODD(53,x)
   if bfp_ABSOLUTE(r) > bfp_NMAX_BFP64 then do
      if FPSCR.OE=0 then do
         if ro=0 & rmode=0b00 then r ← x.sign ? bfp_INFINITY   : bfp_INFINITY
         if ro=0 & rmode=0b01 then r ← x.sign ? bfp_NMAX_BFP64 : bfp_NMAX_BFP64
         if ro=0 & rmode=0b10 then r ← x.sign ? bfp_NMAX_BFP64 : bfp_INFINITY
         if ro=0 & rmode=0b11 then r ← x.sign ? bfp_INFINITY   : bfp_NMAX_BFP64
         if ro=1              then r ← x.sign ? bfp_NMAX_BFP64 : bfp_NMAX_BFP64
         r.sign   ← x.sign
         ox_flag  ← 0b1
         xx_flag  ← 0b1
         inc_flag ← 0bU
         return(r)
      end
      else do
         r.exponent ← r.exponent - 1536
         ox_flag ← 1
      end
   end
   return r   (binary floating-point working format)
```

`bfp_ROUND_TO_BFP64_DEFAULT(rmode, x)`

`x` is a normalized binary floating-point value that is represented in the binary floating-point working format and has unbounded exponent range and significand precision.

`rmode` is a 2-bit integer value specifying one of four rounding modes.

`rmode=0b00` Round to Nearest Even
`rmode=0b01` Round towards Zero
`rmode=0b10` Round towards + Infinity
`rmode=0b11` Round towards - Infinity

If `x` is a QNaN, Infinity, or Zero, return `x`.
Otherwise, if `x` is an SNaN, set `vxsnan_flag` to 1 and return the corresponding QNaN representation of `x`.
Otherwise, return the value `x` rounded to single-precision format's exponent range and significand precision represented in the floating-point working format using the rounding mode specified by `rmode`.
The result is that which would be produced if OE and UE were set to 0 (the "default" value).

```
if x.class.SNaN then do
   vxsnan_flag ← 1
   return bfp_QUIET(x)
end
if x.class.QNaN then return x
if x.class.Infinity then return x
if x.class.Zero then return x
if bfp_ABSOLUTE(x) < bfp_NMIN_BFP64 then do
   x = bfp_DENORM(-1022,x)
   if rmode=0b00 then r = bfp_ROUND_NEAR_EVEN(53,x)
   if rmode=0b01 then r = bfp_ROUND_TRUNC(53,x)
   if rmode=0b10 then r = bfp_ROUND_CEIL(53,x)
   if rmode=0b11 then r = bfp_ROUND_FLOOR(53,x)
  if FPSCR.UE=0 then do
    ux_flag ← xx_flag
  else do
    ux_flag ←  0b1

    return r
end
if rmode=0b00 then r = bfp_ROUND_NEAR_EVEN(53,x)
if rmode=0b01 then r = bfp_ROUND_TRUNC(53,x)
if rmode=0b10 then r = bfp_ROUND_CEIL(53,x)
if rmode=0b11 then r = bfp_ROUND_FLOOR(53,x)
if bfp_ABSOLUTE(r)>bfp_NMAX_BFP32 then do
   if rmode=0b00 then r = x.sign ? bfp_INFINITY   : bfp_INFINITY
   if rmode=0b01 then r = x.sign ? bfp_NMAX_BFP64 : bfp_NMAX_BFP64
   if rmode=0b10 then r = x.sign ? bfp_NMAX_BFP64 : bfp_INFINITY
   if rmode=0b11 then r = x.sign ? bfp_INFINITY   : bfp_NMAX_BFP64
   r.sign   = x.sign
   if FPSCR.OE=0 then do
      ox_flag ← 0b1
      xx_flag ← 0b1
      inc_flag ← 0bU
   else do
      ox_flag ←  0b1
end
return r
```

`bfp_ROUND_TO_BFP80(rmode, x)`

`x` is a normalized binary floating-point value that is represented in the binary floating-point working format and has unbounded exponent range and significand precision.

`rmode` is a 2-bit unsigned integer 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.SNaN then do
   vxsnan_flag ← 1
   return bfp_QUIET(x)
end

if x.class.QNaN     then return x
if x.class.Infinity then return x
if x.class.Zero     then return x
if bfp_ABSOLUTE(x) < bfp_NMIN_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    (binary floating-point working format)
```

bfp_ROUND_TO_BFP128(ro, rmode, x)

x is a normalized binary floating-point value that is represented in the binary floating-point working format and has unbounded exponent range and significand precision.

ro is a 1-bit unsigned integer and rmode is a 2-bit unsigned integer, together specifying one of five rounding modes to be used in rounding z.

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.SNaN then do
         vxsnan_flag ← 1
         return bfp_QUIET(x)
      end

      if x.class.QNaN     then return x
      if x.class.Infinity then return x
      if x.class.Zero     then return x
      if bfp_ABSOLUTE(x) < bfp_NMIN_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_INFINITY
            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  ← 0b1
            xx_flag  ← 0b1
            inc_flag ← 0bU
            return(r)
         end
         else do
            r.exponent ← r.exponent - 24576
            ox_flag ← 1
         end
      end
      return r    (binary floating-point working format)
```

bfp_ROUND_TO_INTEGER(rmode, x)

x is a binary floating-point value that is represented in the binary floating-point working format and has unbounded exponent range and significand precision.

If x is an SNaN, vxsnan_flag is set to 1.

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 towards 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 binary floating-point working format and has unbounded exponent range and significand precision. x must be rounded as presented, without prenormalization.

p is an integer value specifying the precision (i.e., number of bits) the significand is rounded to.

Return x with bit p-1 of the significand set to 1 if any of the bits to the right of bit p-1 of the significand of x are equal to 1, and all bits to the right of bit p-1 of the significand of the value returned are set to 0. Otherwise return x with all bits to the right of bit p-1 of the significand set to 0.

inc_flag is set to 1 if the magnitude of the value returned is greater than x.
xx_flag is set to 1 if the value returned is not equal to x.

bfp_ROUND_NEAR_EVEN(p, x)

x is a binary floating-point value that is represented in the binary floating-point working format and has unbounded exponent range and significand precision. x must be rounded as presented, without prenormalization.

p is an integer value specifying the precision (i.e., number of bits) the significand is rounded to.

Return the floating-point number having unbounded exponent range and a significand with a width of p bits that is nearest in value to x (in case of a tie, the floating-point number having unbounded exponent range and a p-bit significand with the least-significant bit equal to 0 is used).

inc_flag is set to 1 if the magnitude of the value returned is greater than x.
xx_flag is set to 1 if the value returned is not equal to x.

bfp_ROUND_TRUNC(p, x)

x is a binary floating-point value that is represented in the binary floating-point working format and has unbounded exponent range and significand precision. x must be rounded as presented, without prenormalization.

p is an integer value specifying the precision (i.e., number of bits) the significand is rounded to.

Return the largest floating-point number having unbounded exponent range and a significand with a width of p bits that is lesser or equal in value to x if x>0, or the smallest floating-point number having unbounded exponent range but double-precision significand precision that is greater or equal in value to x if x<0.

inc_flag is set to 1 if the magnitude of the value returned is greater than x.
xx_flag is set to 1 if the value returned is not equal to x.

bfp_SQUARE_ROOT(x)

x is a binary floating-point value that is represented in the binary floating-point working format and has unbounded exponent range and significand precision.

If x is an SNaN, vxsnan_flag is set to 1.
Otherwise, if x is negative and non-zero, vxsqrt_flag is set to 1.

If x is a QNaN, return x.
Otherwise, if x is an SNaN, return x represented as a QNaN.
Otherwise, if x is −Zero, return −Zero.
Otherwise, if x is negative, return the standard QNaN.
Otherwise, return the normalized square root of x, represented in the binary floating-point working format, having unbounded range and precision.

bfp16_CONVERT_FROM_BFP(x)

x is a floating-point value represented in the binary floating-point working format.

If x.class.SNaN=1 or x.class.QNaN=1, do the following.
- Bit 0 of result is set to the value of x.sign.
- Bits 1:5 of result are set to the value 0b11111.
- Bits 6:15 of result are set to the value of bits 1:10 of x.significand.

Otherwise, if x.class.Infinity=1, do the following.
- Bit 0 of result is set to the value of x.sign.
- Bits 1:5 of result are set to the value 0b11111.
- Bits 6:15 of result are set to 0.

Otherwise, if x.class.Zero=1, do the following.
- Bit 0 of result is set to the value of x.sign.
- Bits 1:15 of result are set to 0.

Otherwise, if x.exponent is less than -14 and UE=0, do the following.
- Bit 0 of result is set to the value of x.sign.

sh_cnt is set to the difference, -14 - x.exponent.
Bits 1:5 of result are set to 0b00000.
Bits 6:15 of result are set to bits 1:10 of x.significand shifted right by sh_cnt bits.

Otherwise, if x.exponent is less than -14 and UE=1, result is undefined.
Otherwise, if x.exponent is greater than 15 and OE=1, result is undefined.
Otherwise, do the following.
Bit 0 of result is set to the value of x.sign.
Bits 1:5 of result are set to the sum, x.exponent + 15.
Bits 6:15 of result are set to bits 1:10 of x.significand.
Return result.

bfp32_ABSOLUTE(x)
x is a floating-point value represented in single-precision format.
Return x with its sign set to 0.

bfp32_CONVERT_FROM_BFP(x)
x is a floating-point value represented in the binary floating-point working format.
If x.class.SNaN=1 or 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.

bfp32_CONVERT_FROM_BFP64(x)
x is a single-precision floating-point value in double-precision format.
Returns the value x in single-precision format. x must be representable in single-precision, or else 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.

```
sign ← x.bit[0]
exp  ← x.bit[1:11] - 1023
frac ← x.bit[12:63]

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
   msb = 1
   do while (exp < -126)     // denormalize operand until exp=Emin
      frac.bit[1:51] ← frac.bit[0:50]
      frac.bit[0]    ← msb
```

```
            msb            ← 0
            exp            ← exp + 1
         end
         if (frac = 0) then return(0xUUUU_UUUU)  // value not representable in SP format
         else do // return denormal SP
            result.bit[0]    ← sign
            result.bit[1:8]  ← 0
            result.bit[9:31] ← frac.bit[0:22]
            return(result)
         end
      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
         result.bit[0]    ← sign
         result.bit[1:8]  ← 255
         result.bit[9:31] ← frac.bit[0:22]
         return(result)
      end
      else if (exp < +1024) & (exp > +126) then return(0xUUUU_UUUU)  // overflow
      else do  // normal value
         result.bit[0]    ← sign
         result.bit[1:8]  ← exp.bit[4:11] + 127
         result.bit[9:31] ← frac.bit[0:22]
         return(result)
      end
```

bfp64_IS_BFP32_VALUE(x)

x is a double-precision floating-point value.

Returns true if x is a value representable in single-precision format, or false if not.

```
      sign ← x.bit[0]
      exp  ← x.bit[1:11] - 1023
      frac ← x.bit[12:63]

      if      (exp = -1023) & (frac  = 0) then return(true)  // +/- Zero
      else if (exp = -1023) & (frac != 0) then return(false) // DP denorm
      else if (exp < -126) then do // denormalization required
         msb = 1
         do while (exp < -126)     // denormalize operand until exp=Emin
            if frac.bit[51] = 0b1 then return(false)
            frac.bit[1:51] ← frac.bit[0:50]
            frac.bit[0]    ← msb
            msb            ← 0
            exp            ← exp + 1
         end
         return(true)         // denormal SP
      end
      else if (exp <= +126) then return(true)  // normal SP value
      else if (exp < +1024) then return(false) // overflow
      else                       return(true)  // +/- Infinity or NaN
```

bfp32_MAXIMUM(x, y)

x is a binary floating-point value that is represented in single-precision format.

y is a binary floating-point value that is represented in single-precision format.

If x or y is an SNaN, vxsnan_flag is set to 1.

If x is a QNaN and y is not a NaN, return y.

Otherwise, if x is a QNaN, return x.

Otherwise, if x is an SNaN, return x represented as a QNaN.

Otherwise, if y is a QNaN, return x.
Otherwise, if y is an SNaN, return y represented as a QNaN.
Otherwise, return the greater of x and y, where +0 is considered greater than -0.

bfp32_MINIMUM(x, y)
x is a binary floating-point value that is represented in single-precision format.
y is a binary floating-point value that is represented in single-precision format.
If x or y is an SNaN, vxsnan_flag is set to 1.
If x is a QNaN and y is not a NaN, return y.
Otherwise, if x is a QNaN, return x.
Otherwise, if x is an SNaN, return x represented as a QNaN.
Otherwise, if y is a QNaN, return x.
Otherwise, if y is an SNaN, return y represented as a QNaN.
Otherwise, return the lesser of x and y, where -0 is considered less than +0.

bfp32_NEGATE(x)
x is a floating-point value represented in single-precision format.
Return x with its sign complemented.

bfp32_NEGATIVE_ABSOLUTE(x)
x is a floating-point value represented in single-precision format.
Return x with its sign set to 1.

bfp64_ABSOLUTE(x)
x is a floating-point value represented in double-precision format.
Return x with its sign set to 0.

bfp64_CONVERT_FROM_BFP(x)
x is a floating-point value represented in the binary floating-point working format.
If x.class.SNaN=1 or 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.

bfp64_NEGATE(x)
x is a floating-point value represented in double-precision format.
Return x with its sign complemented.

`bfp64_NEGATIVE_ABSOLUTE(x)`

`x` is a floating-point value represented in double-precision format.

Return `x` with its sign set to 1.

`bfp64_MAXIMUM(x, y)`

`x` is a binary floating-point value that is represented in double-precision format.
`y` is a binary floating-point value that is represented in double-precision format.

If x or y is an SNaN, `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.

`bfp64_MAXIMUM_TYPE_C(x, y)`

`x` is a binary floating-point value that is represented in double-precision format.
`y` is a binary floating-point value that is represented in double-precision format.

If `x` or `y` is an SNaN, `vxsnan_flag` is set to 1.

If `x` or `y` is a NaN, return `y`.
Otherwise, if `x` is greater than `y`, return `x`.
Otherwise, return `y`.

`bfp64_MAXIMUM_TYPE_J(x, y)`

`x` is a binary floating-point value that is represented in double-precision format.
`y` is a binary floating-point value that is represented in double-precision format.

If `x` or `y` is an SNaN, `vxsnan_flag` is set to 1.

If `x` is a NaN, return `x`.
Otherwise, if `y` is a NaN, return `y`.
Otherwise, if both `x` and `y` are Zero and either `x` or `y` is a +Zero, return +Zero.
Otherwise, if both `x` and `y` are Zero and both `x` and `y` are -Zero, return -Zero.
Otherwise, if `x` is greater than `y`, return `x`.
Otherwise, return `y`.

`bfp64_MINIMUM(x, y)`

`x` is a binary floating-point value that is represented in double-precision format.
`y` is a binary floating-point value that is represented in double-precision format.

If x or y is an SNaN, `vxsnan_flag` is set to 1.

If x is a QNaN and y is not a NaN, return y.
Otherwise, if x is a QNaN, return x.
Otherwise, if x is an SNaN, return x represented as a QNaN.
Otherwise, if y is a QNaN, return x.
Otherwise, if y is an SNaN, return y represented as a QNaN.
Otherwise, return the lesser of x and y, where -0 is considered less than +0.

`bfp64_MINIMUM_TYPE_C(x, y)`

`x` is a binary floating-point value that is represented in double-precision format.
`y` is a binary floating-point value that is represented in double-precision format.

If `x` or `y` is an SNaN, `vxsnan_flag` is set to 1.

If `x` or `y` is a NaN, return `y`.
Otherwise, if `x` is less than `y`, return `x`.
Otherwise, return `y`.

`bfp64_MINIMUM_TYPE_J(x, y)`

`x` is a binary floating-point value that is represented in double-precision format.
`y` is a binary floating-point value that is represented in double-precision format.

If `x` or `y` is an SNaN, `vxsnan_flag` is set to 1.

If x is a NaN, return x.
Otherwise, if y is a NaN, return y.
Otherwise, if both x and y are Zero and either x or y is a -Zero, return -Zero.
Otherwise, if both x and y are Zero and both x and y are +Zero, return +Zero.
Otherwise, if x is less than y, return x.
Otherwise, return y.

bfp128_ABSOLUTE(x)
x is a floating-point value represented in quad-precision format.
Return x with its sign set to 0.

bfp128_CONVERT_FROM_BFP(x)
x is a quad-precision floating-point value that is represented in the binary floating-point working format.
If x is an SNaN or a QNaN,
the contents of bit 0 of result are set to the value of x.sign,
the contents of bits 1:15 of result are set to the value 0b111_1111_1111_1111, and
the contents of bits 16:127 of result are set to the value of bits 1:112 of x.significand.
Otherwise, if x is a Zero,
the contents of bit 0 of result are set to the value of x.sign, and
the contents of bits 1:15 of result are set to the value 0b000_0000_0000_0000, and
the contents of bits 16:127 of result are set to the value 0x0000_0000_0000_0000_0000_0000_0000.
Otherwise, if x is an Infinity,
the contents of bit 0 of result are set to the value of x.sign,
the contents of bits 1:15 of result are set to the value 0b111_1111_1111_1111, and
the contents of bits 16:127 of result are set to the value 0x0000_0000_0000_0000_0000_0000_0000.
Otherwise, do the following.
If the exponent of x is less than -16382,
the contents of bit 0 of result are set to the value of x.sign,
the contents of bits 1:15 of result are set to the value 0b000_0000_0000_0000, and
the contents of bits 16:127 of result are set to the value of bits 1:112 of the significand of x shifted right by N bits, where N is the value -16382 subtracted by the value of the exponent of x.
Otherwise,
the contents of bit 0 of result are set to the value of x.sign,
the contents of bits 1:15 of result are set to the sum of the exponent of x and 16383, and
the contents of bits 16:127 of result are set to the value of bits 1:112 of the significand of x.
Return result (i.e., x in quad-precision format).

bfp128_NEGATE(x)
x is a floating-point value represented in quad-precision format.
Return x with its sign complemented.

bfp128_NEGATIVE_ABSOLUTE(x)
x is a floating-point value represented in quad-precision format.
Return x with its sign set to 1.

si64_CONVERT_FROM_BFP(x)
x is an integer value represented in the binary floating-point working format.
Return the value x in signed doubleword integer format.

ui64_CONVERT_FROM_BFP(x)
x is an integer value represented in the binary floating-point working format.
Return the value x in 64-bit unsigned integer format.

bfp128_MAXIMUM_TYPE_C(x, y)
x is a binary floating-point value that is represented in quad-precision format.
y is a binary floating-point value that is represented in quad-precision format.
If x or y is an SNaN, vxsnan_flag is set to 1.
If x or y is a NaN, return y.
Otherwise, if x is greater than y, return x.
Otherwise, return y.

bfp128_MAXIMUM_TYPE_J(x, y)
x is a binary floating-point value that is represented in quad-precision format.
y is a binary floating-point value that is represented in quad-precision format.
If x or y is an SNaN, vxsnan_flag is set to 1.
If x is a NaN, return x.
Otherwise, if y is a NaN, return y.
Otherwise, if both x and y are Zero and either x or y is a +Zero, return +Zero.
Otherwise, if both x and y are Zero and both x and y are -Zero, return -Zero.
Otherwise, if x is greater than y, return x.
Otherwise, return y.

bfp128_MINIMUM_TYPE_C(x, y)
x is a binary floating-point value that is represented in quad-precision format.
y is a binary floating-point value that is represented in quad-precision format.
If x or y is an SNaN, vxsnan_flag is set to 1.
If x or y is a NaN, return y.
Otherwise, if x is less than y, return x.
Otherwise, return y.

bfp128_MINIMUM_TYPE_J(x, y)
x is a binary floating-point value that is represented in quad-precision format.
y is a binary floating-point value that is represented in quad-precision format.
If x or y is an SNaN, vxsnan_flag is set to 1.
If x is a NaN, return x.
Otherwise, if y is a NaN, return y.
Otherwise, if both x and y are Zero and either x or y is a -Zero, return -Zero.
Otherwise, if both x and y are Zero and both x and y are +Zero, return +Zero.
Otherwise, if x is less than y, return x.
Otherwise, return y.

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 as represented in half-precision format.
Return `0b10100` if `x` is a positive denormal value as represented in half-precision format.
Return `0b01000` if `x` is a negative normal value as represented in half-precision format.
Return `0b00100` if `x` is a positive normal value as represented in half-precision format.

`fprf_CLASS_BFP32(x)`

`x` is a floating-point value represented in single-precision format.

Return the 5-bit code that specifies the sign and class of `x`.

Return `0b10001` if `x` is a Quiet NaN.
Return `0b01001` if `x` is a negative infinity.
Return `0b00101` if `x` is a positive infinity.
Return `0b10010` if `x` is a negative zero.
Return `0b00010` if `x` is a positive zero.
Return `0b11000` if `x` is a negative denormal value as represented in single-precision format.
Return `0b10100` if `x` is a positive denormal value as represented in single-precision format.
Return `0b01000` if `x` is a negative normal value as represented in single-precision format.
Return `0b00100` if `x` is a positive normal value as represented in single-precision format.

`fprf_CLASS_BFP64(x)`

`x` is a floating-point value represented in double-precision format.

Return the 5-bit code that specifies the sign and class of `x`.

Return `0b10001` if `x` is a Quiet NaN.
Return `0b01001` if `x` is a negative infinity.
Return `0b00101` if `x` is a positive infinity.
Return `0b10010` if `x` is a negative zero.
Return `0b00010` if `x` is a positive zero.
Return `0b11000` if `x` is a negative denormal value as represented in double-precision format.
Return `0b10100` if `x` is a positive denormal value as represented in double-precision format.
Return `0b01000` if `x` is a negative normal value as represented in double-precision format.
Return `0b00100` if `x` is a positive normal value as represented in double-precision format.

`fprf_CLASS_BFP128(x)`

`x` is binary floating-point value that is represented in quad-precision format.

Return the 5-bit characterization of the sign and class of `x`.

Return `0b10001` if `x` is a Quiet NaN.
Return `0b01001` if `x` is negative and an infinity.
Return `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.

`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.
`vxzdz_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.

`SetFX(x)`
x is one of the exception flags in the FPSCR.
If the contents of x is 0, `FX` and x are set to 1.

`si128_CONVERT_FROM_BFP(x)`
`x` is an integer value represented in the binary floating-point working format.
If `x` is a NaN,
`vxcvi_flag` is set to 1,
`vxsnan_flag` is set to 1 if `x` is an SNaN, and
return `0x8000_0000_0000_0000_0000_0000_0000_0000`,
Otherwise, do the following.
Let `rnd` be the value `x` truncated to an integral value.

Let `exponent` be the unbiased exponent of `rnd`.
Let `significand` be the significand of `rnd`.

If `rnd` is greater than $2^{127} - 1$,
`vxcvi_flag` is set to 1, and
return `0x7FFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF`.

Otherwise, if `rnd` is less than -2^{127},
`vxcvi_flag` is set to 1, and
return `0x8000_0000_0000_0000_0000_0000_0000_0000`.

Otherwise,
`xx_flag` is set to 1 if `rnd` is inexact,
`inc_flag` is set to 0,
`significand` is shifted right by the difference 127-`exponent` with 0s shifted in,
if `rnd` is negative, `significand` is negated, and
return bits 0:127 of `significand`.

`si32_CHOP(x)`
`x` is a signed integer value.
Return the rightmost 32 bits of x in 32-bit signed integer format.

si32_CLAMP(x)

x is a signed integer value.

If x is greater than $2^{31} - 1$, result is the value $2^{31} - 1$, and SAT is set to 1.

Otherwise, if x is less than -2^{31}, result is the value -2^{31}, and SAT is set to 1.

Otherwise, result is x.

Return x in 32-bit signed integer format.

ui128_CONVERT_FROM_BFP(x)

x is an integer value represented in the binary floating-point working format.

If x is a NaN,

vxcvi_flag is set to 1,

vxsnan_flag is set to 1 if x is an SNaN, and

return 0x0000_0000_0000_0000_0000_0000_0000_0000.

Otherwise, do the following.

Let rnd be the value x truncated to an integral value.

Let exponent be the unbiased exponent of rnd.

Let significand be the significand of rnd.

If rnd is greater than $2^{128} - 1$,

vxcvi_flag is set to 1, and

return 0xFFFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF.

Otherwise, if rnd is less than 0,

vxcvi_flag is set to 1, and

return 0x0000_0000_0000_0000_0000_0000_0000_0000.

Otherwise,

xx_flag is set to 1 if rnd is inexact,

inc_flag is set to 0,

significand is shifted right by the difference 127−exponent with 0s shifted in, and

return bits 0:127 of significand.

7.6.2 VSX Instruction Descriptions

7.6.2.1 VSX Storage Access Instructions

Load VSX Scalar instructions place a copy of the contents of the addressed byte, halfword, word, or doubleword in storage into the left-most doubleword element of the target VSR. For integer byte, halfword, and word forms, the data are placed into the rightmost byte, halfword, or word of the doubleword, and the leftmost bits of the doubleword are set to 0 (or set to the copy of the sign bit for ***lxsiwax***). For the single-precision floating-point word form, the data is converted to double-precision format and placed into the doubleword. The contents of the right-most doubleword element of the target VSR are set to 0.

Store VSX Scalar instructions place a copy of the contents of the leftmost doubleword element (or portions of) in the source VSR into the addressed byte, halfword, word or doubleword in storage. For integer byte, halfword, and word forms, the rightmost byte, halfword, or word of the doubleword are stored.

Load VSX Vector instructions load a quadword from storage as a vector of 16 byte elements, 8 halfword elements, 4 word elements, 2 doubleword elements or a quadword element into a VSR.

Load VSX Vector & Splat instructions load a word or doubleword from storage and replicate the data into the 4 words or 2 doublewords of a VSR.

Store VSX Vector instructions store a vector of 16 byte elements, 8 halfword elements, 4 word elements, 2 doubleword elements or a quadword element from a VSR into a quadword in storage.

Load VSX Vector with Length instruction loads from 0 to 16 bytes into a VSR.

Store VSX Vector with Length instruction stores from 0 to 16 bytes from a VSR.

Load VSX Vector Paired instructions load an octword (32 bytes) from storage into two sequential VSRs (i.e., a vector of 32 byte elements, 16 halfword elements, 8 word elements, 4 doubleword elements or 2 quadword elements).

VSX Vector Store Paired instructions store the contents of two sequential VSRs into an octword (32 bytes) in storage (i.e., a vector of 32 byte elements, 16 halfword elements, 8 word elements, 4 doubleword elements or 2 quadword elements).

7.6.2.1.1 VSX Scalar Storage Access Instructions

Load VSX Scalar Doubleword DS-form

lxsd VRT,disp(RA)

57	VRT	RA	DS	2
0	6	11	16	30 31

Prefixed Load VSX Scalar Doubleword 8LS:D-form

plxsd VRT,D(RA),R

Prefix:

1	0	0	//	R	//	d0
0	6	8	9	11	12	14 31

Suffix:

42	VRT	RA	d1
0	6	11	16 31

```
if MSR.VSX=0 then VSX_Unavailable()

if ''lxsd''  then
   EA ← (RA|0) + EXTS64(DS||0b00)
if ''plxsd'' & R=0 then
   EA ← (RA|0) + EXTS64(d0||d1)
if ''plxsd'' & R=1 then
   EA ← CIA + EXTS64(d0||d1)

VSR[VRT+32].dword[0] ← MEM(EA,8)
VSR[VRT+32].dword[1] ← 0x0000_0000_0000_0000
```

Let XT be the value VRT + 32.

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

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

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

When Big-Endian byte ordering is employed, the contents of the doubleword in storage at address EA are placed into 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[VRT+32].

The contents of doubleword element 1 of VSR[VRT+32] are set to 0.

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

Special Registers Altered:

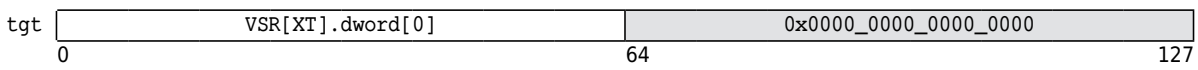
None

Extended Mnemonics:

Extended Mnemonics for *Prefixed Load VSX Scalar Doubleword*:

Extended mnemonic:		Equivalent to:	
plxsd	VRT,D(RA)	plxsd	VRT,D(RA),0
plxsd	VRT,D	plxsd	VRT,D(0),1

VSR Data Layout for [p]lxsd



Load VSX Scalar Doubleword Indexed X-form

lxsdx XT,RA,RB

31	T	RA	RB	588	TX
0	6	11	16	21	31

```
if MSR.VSX=0 then VSX_Unavailable()

EA ← ((RA=0) ? 0 : GPR[RA]) + GPR[RB]

VSR[32×TX+T].dword[0] ← MEM(EA,8)
VSR[32×TX+T].dword[1] ← 0x0000_0000_0000_0000
```

Let XT be the value 32×TX + T.

Let EA be the sum of the contents of GPR[RA], or 0 if RA is equal to 0, and the contents of GPR[RB].

When Big-Endian byte ordering is employed, the contents of the doubleword in storage at address EA are placed into load_data in such an order that;

- the contents of the byte in storage at address EA are placed into byte 0 of load_data,
- the contents of the byte in storage at address EA+1 are placed into byte 1 of load_data, and so forth until
- the contents of the byte in storage at address EA+7 are placed into byte 7 of load_data.

When Little-Endian byte ordering is employed, the contents of the doubleword in storage at address EA are placed into load_data in such an order that;

- the contents of the byte in storage at address EA are placed into byte 7 of load_data,
- the contents of the byte in storage at address EA+1 are placed into byte 6 of load_data, and so forth until
- the contents of the byte in storage at address EA+7 are placed into byte 0 of load_data.

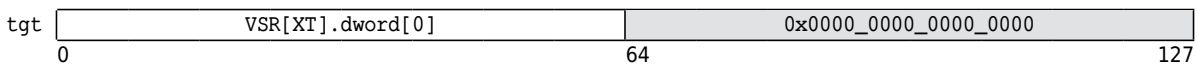
load_data is placed into doubleword element 0 of VSR[XT].

The contents of doubleword element 1 of VSR[XT] are set to 0.

Special Registers Altered:

None

VSR Data Layout for lxsdx



Load VSX Scalar as Integer Byte & Zero Indexed X-form

lxsibzx XT,RA,RB

31	T	RA	RB	781	TX
0	6	11	16	21	31

```
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] ← 0x0000_0000_0000_0000
```

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 set to 0.

Special Registers Altered:

None

Load VSX Scalar as Integer Halfword & Zero Indexed X-form

lxsihzx XT,RA,RB

31	T	RA	RB	813	TX
0	6	11	16	21	31

```
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] ← 0x0000_0000_0000_0000
```

Let XT be the value 32×TX + T.

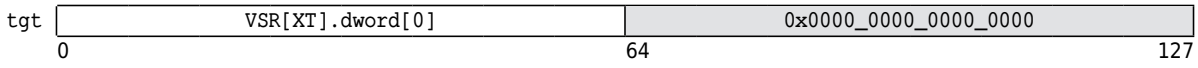
Let the effective address (EA) be sum of the contents of GPR[RA], or 0 if RA is equal to 0, and the contents of GPR[RB].

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 set to 0.

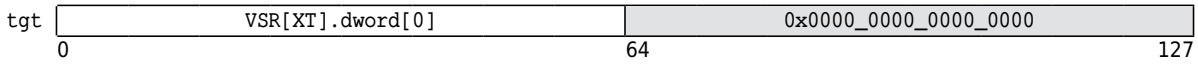
Special Registers Altered:

None

VSR Data Layout for lxsibzx



VSR Data Layout for lxsihzx



Load VSX Scalar as Integer Word Algebraic Indexed X-form

lxsiwax XT,RA,RB

31	T	RA	RB	76	TX
0	6	11	16	21	31

```
if MSR.VSX=0 then VSX_Unavailable()

EA ← ((RA=0) ? 0 : GPR[RA]) + GPR[RB]

VSR[32×TX+T].dword[0] ← EXTS64(MEM(EA,4))
VSR[32×TX+T].dword[1] ← 0x0000_0000_0000_0000
```

Let XT be the value 32×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 sign-extended to a doubleword and placed in doubleword element 0 of VSR[XT].

The contents of doubleword element 1 of VSR[XT] are set to 0.

Special Registers Altered:

None

VSR Data Layout for lxsiwax

Load VSX Scalar as Integer Word & Zero Indexed X-form

lxsiwzx XT,RA,RB

31	T	RA	RB	12	TX
0	6	11	16	21	31

```
if MSR.VSX=0 then VSX_Unavailable()

EA ← ((RA=0) ? 0 : GPR[RA]) + GPR[RB]

VSR[32×TX+T].dword[0] ← ExtendZero(MEM(EA,4))
VSR[32×TX+T].dword[1] ← 0x0000_0000_0000_0000
```

Let XT be the value 32×TX + T.

Let EA be the sum of the contents of GPR[RA], or 0 if RA is equal to 0, and the contents of GPR[RB].

When Big-Endian byte ordering is employed, the contents of the word in storage at address EA are placed into load_data in such an order that;

- the contents of the byte in storage at address EA are placed into byte 0 of load_data,
- the contents of the byte in storage at address EA+1 are placed into byte 1 of load_data,
- the contents of the byte in storage at address EA+2 are placed into byte 2 of load_data, and
- the contents of the byte in storage at address EA+3 are placed into byte 3 of load_data.

When Little-Endian byte ordering is employed, the contents of the word in storage at address EA are placed into load_data in such an order that;

- the contents of the byte in storage at address EA are placed into byte 3 of load_data,
- the contents of the byte in storage at address EA+1 are placed into byte 2 of load_data,
- the contents of the byte in storage at address EA+2 are placed into byte 1 of load_data, and
- the contents of the byte in storage at address EA+3 are placed into byte 0 of load_data.

load_data is zero-extended and placed in doubleword element 0 of VSR[XT].

The contents of doubleword element 1 of VSR[XT] are set to 0.

Special Registers Altered:

None

VSR Data Layout for lxsiwzx

Load VSX Scalar Single-Precision DS-form

lxssp VRT,disp(RA)

57	VRT	RA	DS	3
0	6	11	16	30 31

Prefixed Load VSX Scalar Single-Precision 8LS:D-form

plxssp VRT,D(RA),R

Prefix:

1	0	0	//	R	//	d0
0	6	8	9	11	12	14 31

Suffix:

43	VRT	RA	d1
0	6	11	16 31

```
if MSR.VEC=0 then Vector_Unavailable()

if ''lxssp''  then
   EA ← (RA|0) + EXTS64(DS||0b00)
if ''plxssp'' & R=0 then
   EA ← (RA|0) + EXTS64(d0||d1)
if ''plxssp'' & R=1 then
   EA ← CIA + EXTS64(d0||d1)

load_data ← MEM(EA,4)
result ← bfp_CONVERT_FROM_BFP32(MEM(EA,4))
VSR[VRT+32].dword[0] ← bfp64_CONVERT_FROM_BFP(result)
VSR[VRT+32].dword[1] ← 0x0000_0000_0000_0000
```

Let XT be the value VRT + 32.

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

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

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

When Big-Endian byte ordering is employed, the contents of the 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, 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 set to 0.

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

Special Registers Altered:

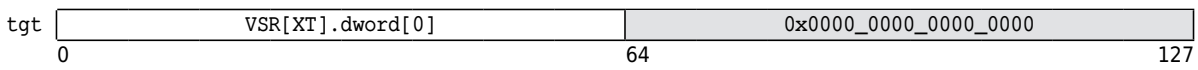
None

Extended Mnemonics:

Extended Mnemonics for *Prefixed Load VSX Scalar Single*:

Extended mnemonic:		Equivalent to:	
plxssp	VRT,D(RA)	plxssp	VRT,D(RA),0
plxssp	VRT,D	plxssp	VRT,D(0),1

VSR Data Layout for [p]lxssp



Load VSX Scalar Single-Precision Indexed X-form

lxsspx XT,RA,RB

31	T	RA	RB	524	TX
0	6	11	16	21	31

```
if MSR.VSX=0 then VSX_Unavailable()

EA ← ((RA=0) ? 0 : GPR[RA]) + GPR[RB]

load_data ← MEM(EA,4)
result ← bfp_CONVERT_FROM_BFP32(MEM(EA,4))
VSR[VRT+32].dword[0] ← bfp64_CONVERT_FROM_BFP(result)
VSR[VRT+32].dword[1] ← 0x0000_0000_0000_0000
```

Let XT be the value 32×TX + T.

Let EA be the sum of the contents of GPR[RA], or 0 if RA is equal to 0, and the contents of GPR[RB].

When Big-Endian byte ordering is employed, the contents of the word in storage at address EA are placed into 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.

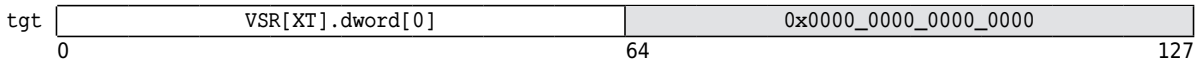
load_data, interpreted as a single-precision floating-point value, is placed in doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are set to 0.

Special Registers Altered:

None

VSR Data Layout for lxsspx



Store VSX Scalar Doubleword DS-form

stxsd VRS,disp(RA)

61	VRS	RA	DS	2
0	6	11	16	30 31

Prefixed Store VSX Scalar Doubleword 8LS:D-form

pstxsd VRS,D(RA),R

Prefix:

1	0	0	//	R	//	d0
0	6	8	9	11	12	14 31

Suffix:

46	VRS	RA	d1
0	6	11	16 31

```
if MSR.VEC=0 then Vector_Unavailable()

if ''stxsd''  then
   EA ← (RA|0) + EXTS64(DS||0b00)
if ''pstxsd'' & R=0 then
   EA ← (RA|0) + EXTS64(d0||d1)
if ''pstxsd'' & R=1 then
   EA ← CIA + EXTS64(d0||d1)

MEM(EA,8)     ← VSR[VRS+32].dword[0]
```

Let XS be the value VRS + 32.

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

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

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

Let store_data be the contents of doubleword element 0 of VSR[XS].

When Big-Endian byte ordering is employed, store_data is placed in the doubleword in storage at address EA in such order that;

- byte 0 of store_data is placed into the byte in storage at address EA,
- byte 1 of store_data is placed into the byte in storage at address EA+1, and so forth until
- byte 7 of store_data is placed into the byte in storage at address EA+7.

When Little-Endian byte ordering is employed, store_data is placed in the doubleword in storage at address EA in such order that;

- the contents of byte 7 of doubleword element 0 of VSR[VRS+32] are placed into the byte in storage at address EA,
- the contents of byte 6 of doubleword element 0 of VSR[VRS+32] are placed into the byte in storage at address EA+1, and so forth until
- the contents of byte 0 of doubleword element 0 of VSR[VRS+32] are placed into the byte in storage at address EA+7.

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

Special Registers Altered:

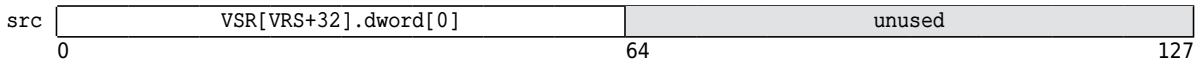
None

Extended Mnemonics:

Extended Mnemonics for *Prefixed Store VSX Scalar Doubleword*:

Extended mnemonic:		Equivalent to:	
pstxsd	VRS,D(RA)	pstxsd	VRS,D(RA),0
pstxsd	VRS,D	pstxsd	VRS,D(0),1

VSR Data Layout for [p]stxsd



Store VSX Scalar Doubleword Indexed X-form

stxsdx XS,RA,RB

31	S	RA	RB	716	SX
0	6	11	16	21	31

```
if MSR.VSX=0 then VSX_Unavailable()

EA ← ((RA=0) ? 0 : GPR[RA]) + GPR[RB]

MEM(EA,8)    ← VSR[XS].dword[0]
```

Let XS be the value 32×SX + S.

Let EA be the sum of the contents of GPR[RA], or 0 if RA is equal to 0, and the contents of GPR[RB].

Let store_data be the contents of doubleword element 0 of VSR[XS].

When Big-Endian byte ordering is employed, store_data is placed in the doubleword in storage at address EA in such order that;

- byte 0 of store_data is placed into the byte in storage at address EA,
- byte 1 of store_data is placed into the byte in storage at address EA+1, and so forth until
- byte 7 of store_data is placed into the byte in storage at address EA+7.

When Little-Endian byte ordering is employed, store_data is placed in the doubleword in storage at address EA in such order that;

- byte 0 of store_data is placed into the byte in storage at address EA+7,
- byte 1 of store_data is placed into the byte in storage at address EA+6, and so forth until
- byte 7 of store_data is placed into the byte in storage at address EA.

Special Registers Altered:

None

VSR Data Layout for stxsdx

Store VSX Scalar as Integer Byte Indexed X-form

stxsibx XS,RA,RB

31	S	RA	RB	909	SX
0	6	11	16	21	31

```
if SX=0 & MSR.VSX=0 then VSX_Unavailable()
if SX=1 & MSR.VEC=0 then Vector_Unavailable()

EA ← ((RA=0) ? 0 : GPR[RA]) + GPR[RB]

MEM(EA,1) ← VSR[32×SX+S].byte[7]
```

Let XS be the value 32×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

Store VSX Scalar as Integer Halfword Indexed X-form

stxsihx XS,RA,RB

31	S	RA	RB	941	SX
0	6	11	16	21	31

```
if SX=0 & MSR.VSX=0 then VSX_Unavailable()
if SX=1 & MSR.VEC=0 then Vector_Unavailable()

EA ← ((RA=0) ? 0 : GPR[RA]) + GPR[RB]

MEM(EA,2) ← VSR[32×SX+S].hword[3]
```

Let XS be the value 32×SX + S.

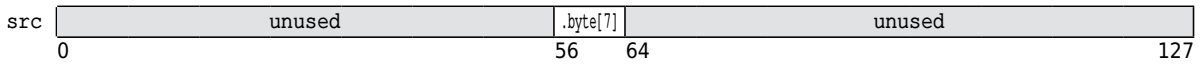
Let the effective address (EA) be sum of the contents of GPR[RA], or 0 if RA is equal to 0, and the contents of GPR[RB].

The contents of halfword element 3 of VSR[XS] are placed into the halfword in storage addressed by EA.

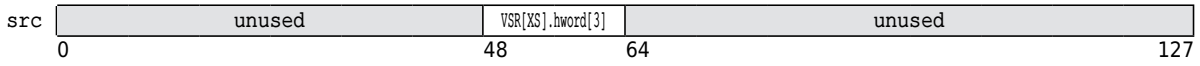
Special Registers Altered:

None

VSR Data Layout for stxsibx



VSR Data Layout for stxsihx



Store VSX Scalar as Integer Word Indexed X-form

stxsiwx XS,RA,RB

31	S	RA	RB	140	SX
0	6	11	16	21	31

```
if MSR.VSX=0 then VSX_Unavailable()

EA ← ((RA=0) ? 0 : GPR[RA]) + GPR[RB]

MEM(EA,4) ← VSR[32×SX+S].word[1]
```

Let XS be the value 32×SX + S.

Let EA be the sum of the contents of GPR[RA], or 0 if RA is equal to 0, and the contents of GPR[RB].

Let store_data be the contents of word element 1 of VSR[XS].

When Big-Endian byte ordering is employed, store_data is placed in the word in storage at address EA in such order that;

- byte 0 of store_data is placed into the byte in storage at address EA,
- byte 1 of store_data is placed into the byte in storage at address EA+1,
- byte 2 of store_data is placed into the byte in storage at address EA+2, and
- byte 3 of store_data is placed into the byte in storage at address EA+3.

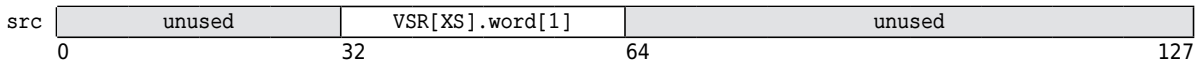
When Little-Endian byte ordering is employed, store_data is placed in the word in storage at address EA in such order that;

- byte 0 of store_data is placed into the byte in storage at address EA+3,
- byte 1 of store_data is placed into the byte in storage at address EA+2,
- byte 2 of store_data is placed into the byte in storage at address EA+1, and
- byte 3 of store_data is placed into the byte in storage at address EA.

Special Registers Altered:

None

VSR Data Layout for stxsiwx



Store VSX Scalar Single-Precision DS-form

stxssp VRS,disp(RA)

61	VRS	RA	DS	3
0	6	11	16	30 31

Prefixed Store VSX Scalar Single-Precision 8LS:D-form

pstxssp VRS,D(RA),R

Prefix:

1	0	0	//	R	//	d0
0	6	8	9	11	12	14 31

Suffix:

47	VRS	RA	d1
0	6	11	16 31

```
if MSR.VEC=0 then Vector_Unavailable()

if ''stxssp''  then
   EA ← (RA|0) + EXTS64(DS||0b00)
if ''pstxssp'' & R=0 then
   EA ← (RA|0) + EXTS64(d0||d1)
if ''pstxssp'' & R=1 then
   EA ← CIA + EXTS64(d0||d1)

MEM(EA,4) ←
   bfp32_CONVERT_FROM_BFP64(VSR[VRS+32].dword[0])
```

Let XS be the value VRS + 32.

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

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

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

Let store_data be the double-precision floating-point value in doubleword element 0 of VSR[XS] converted to single-precision format.

When Big-Endian byte ordering is employed, store_data is placed in the word in storage at address EA in such order that;

- byte 0 of store_data is placed into the byte in storage at address EA,
- byte 1 of store_data is placed into the byte in storage at address EA+1,
- byte 2 of store_data is placed into the byte in storage at address EA+2, and
- byte 3 of store_data is placed into the byte in storage at address EA+3.

When Little-Endian byte ordering is employed, store_data is placed in the word in storage at address EA in such order that;

- byte 0 of store_data is placed into the byte in storage at address EA+3,
- byte 1 of store_data is placed into the byte in storage at address EA+2,
- byte 2 of store_data is placed into the byte in storage at address EA+1, and
- byte 3 of store_data is placed into the byte in storage at address EA.

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

Special Registers Altered:

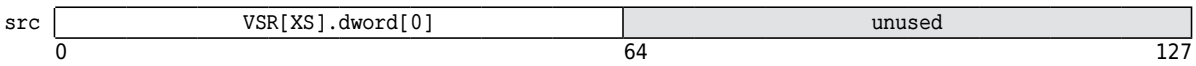
None

Extended Mnemonics:

Extended Mnemonics for *Prefixed Store VSX Scalar Single-Precision*:

Extended mnemonic:	Equivalent to:
pstxssp VRS,D(RA)	pstxssp VRS,D(RA),0
pstxssp VRS,D	pstxssp VRS,D(0),1

VSR Data Layout for [p]stxssp



Store VSX Scalar Single-Precision Indexed X-form

stxsspx XS,RA,RB

31	S	RA	RB	652	SX
0	6	11	16	21	31

```
if MSR.VSX=0 then VSX_Unavailable()

EA ← ((RA=0) ? 0 : GPR[RA]) + GPR[RB]

MEM(EA,4) ←
   bfp32_CONVERT_FROM_BFP64(VSR[32×SX+S].dword[0])
```

Let XS be the value 32×SX + S.

Let EA be the sum of the contents of GPR[RA], or 0 if RA is equal to 0, and the contents of GPR[RB].

Let store_data be the double-precision floating-point value in doubleword element 0 of VSR[XS] converted to single-precision format

When Big-Endian byte ordering is employed, store_data is placed in the word in storage at address EA in such order that;

- byte 0 of store_data is placed into the byte in storage at address EA,
- byte 1 of store_data is placed into the byte in storage at address EA+1,
- byte 2 of store_data is placed into the byte in storage at address EA+2, and
- byte 3 of store_data is placed into the byte in storage at address EA+3.

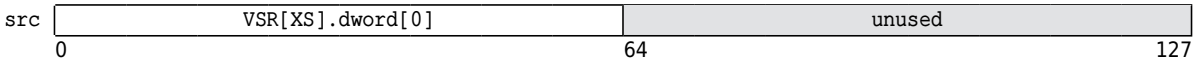
When Little-Endian byte ordering is employed, store_data is placed in the word in storage at address EA in such order that;

- byte 0 of store_data is placed into the byte in storage at address EA+3,
- byte 1 of store_data is placed into the byte in storage at address EA+2,
- byte 2 of store_data is placed into the byte in storage at address EA+1, and
- byte 3 of store_data is placed into the byte in storage at address EA.

Special Registers Altered:

None

VSR Data Layout for stxsspx



7.6.2.1.2 VSX Vector Storage Access Instructions

Load VSX Vector DQ-form

lxv XT,disp(RA)

61	T	RA	DQ	TX	1
0	6	11	16	28	29 31

Prefixed Load VSX Vector 8LS:D-form

plxv XT,D(RA),R

Prefix:

1	0	0	//	R	//	d0
0	6	8	9	11	12 14	31

Suffix:

25	TX	T	RA	d1
0	5	6	11	16 31

```
if ''lxv'' & TX=0 & MSR.VSX=0 then VSX_Unavailable()
if ''lxv'' & TX=1 & MSR.VEC=0 then Vector_Unavailable()
if ''plxv'' & MSR.VSX=0 then VSX_Unavailable()

if ''lxv''  then
   EA ← (RA|0) + EXTS64(DQ||0b0000)
if ''plxv'' & R=0 then
   EA ← (RA|0) + EXTS64(d0||d1)
if ''plxv'' & R=1 then
   EA ← CIA + EXTS64(d0||d1)

VSR[32×TX+T] ← MEM(EA,16)
```

Let XT be the value 32×TX + T.

For ***lxv***, let the effective address (EA) be the sum of the contents of register RA, or the value 0 if RA=0, and the value DQ||0b0000, sign-extended to 64 bits.

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

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

When Big-Endian byte ordering is employed, the contents of the quadword in storage at address EA are placed into load_data in such an order that;

- the contents of the byte in storage at address EA are placed into byte element 0 of load_data,
- the contents of the byte in storage at address EA+1 are placed into byte element 1 of load_data, and so forth until
- the contents of the byte in storage at address EA+15 are placed into byte element 15 of load_data.

When Little-Endian byte ordering is employed, the contents of the quadword in storage at address EA are placed into load_data in such an order that;

- the contents of the byte in storage at address EA are placed into byte element 15 of load_data,
- the contents of the byte in storage at address EA+1 are placed into byte element 14 of load_data, and so forth until
- the contents of the byte in storage at address EA+15 are placed into byte element 0 of load_data.

load_data is placed into VSR[XT].

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

Special Registers Altered:

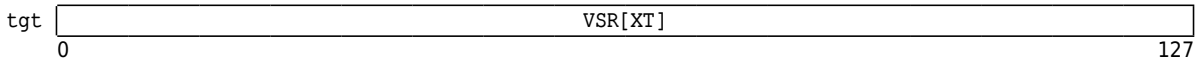
None

Extended Mnemonics:

Extended Mnemonics for *Prefixed Load VSX Vector*:

Extended mnemonic:		Equivalent to:	
plxv	XT,D(RA)	plxv	XT,D(RA),0
plxv	XT,D	plxv	XT,D(0),1

VSR Data Layout for [p]lxv



*Load VSX Vector Byte*16 Indexed X-form*

lxvb16x XT,RA,RB

31	T	RA	RB	876	TX
0	6	11	16	21	31

```
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

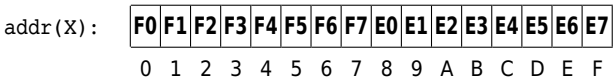
Programming Note

lxvd2x, ***lxvw4x***, ***lxvh8x***, ***lxvb16x***, and ***lxvx*** exhibit identical behavior in Big-Endian mode.

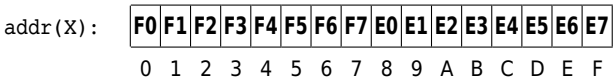
Example: Loading data using *Load VSX Vector Byte*16 Indexed*

```
char  X[] = { 0xF0, 0xF1, 0xF2, 0xF3,
              0xF4, 0xF5, 0xF6, 0xF7,
              0xE0, 0xE1, 0xE2, 0xE3,
              0xE4, 0xE5, 0xE6, 0xE7 };
```

Big-endian storage image of X

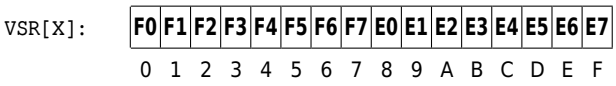


Little-endian storage image of X



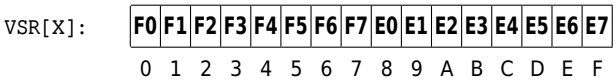
Loading a vector of 16 byte elements from Big-Endian storage in VSR[XT] using ***lxvb16x***, retaining left-to-right element ordering.

```
# Assumptions
#   GPR[PX] = address of X
lxvb16x   xX,r0,rPX
```

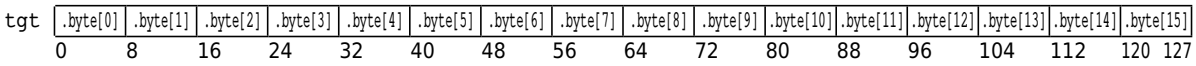


Loading a vector of 16 byte elements from Little-Endian storage in VSR[XT] using ***lxvb16x***, retaining left-to-right element ordering.

```
# Assumptions
#   GPR[PX] = address of X
lxvb16x   xX,r0,rPX
```



VSR Data Layout for lxvb16x



*Load VSX Vector Doubleword*2 Indexed X-form*

lxvd2x XT,RA,RB

31	T	RA	RB	844	TX
0	6	11	16	21	31

```
if MSR.VSX=0 then VSX_Unavailable()

EA ← ((RA=0) ? 0 : GPR[RA]) + GPR[RB]

VSR[32×TX+T].dword[0] ← MEM(EA, 8)
VSR[32×TX+T].dword[1] ← MEM(EA+8, 8)
```

Let XT be the value 32×TX + T.

Let EA be the sum of the contents of GPR[RA], or 0 if RA is equal to 0, and the contents of GPR[RB].

For each integer value i from 0 to 1, do the following.

When Big-Endian byte ordering is employed, the contents of the doubleword in storage at address EA+8×i are placed into load_data in such an order that;

- the contents of the byte in storage at address EA+8×i are placed into byte element 0 of load_data,
- the contents of the byte in storage at address EA+8×i+1 are placed into byte element 1 of load_data, and so forth until
- the contents of the byte in storage at address EA+8×i+7 are placed into byte element 7 of load_data.

When Little-Endian byte ordering is employed, the contents of the doubleword in storage at address EA+8×i are placed into load_data in such an order that;

- the contents of the byte in storage at address EA+8×i are placed into byte element 7 of load_data,
- the contents of the byte in storage at address EA+8×i+1 are placed into byte element 6 of load_data, and so forth until
- the contents of the byte in storage at address EA+8×i+7 are placed into byte element 0 of load_data.

load_data is placed into doubleword element i of VSR[XT].

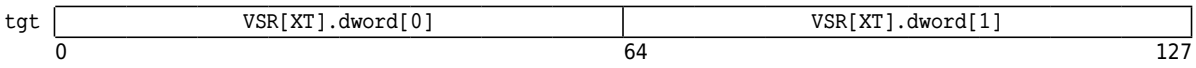
Special Registers Altered:

None

Programming Note

lxvd2x, ***lxvw4x***, ***lxvh8x***, ***lxvb16x***, and ***lxvx*** exhibit identical behavior in Big-Endian mode.

VSR Data Layout for lxvd2x



*Load VSX Vector Halfword*8 Indexed X-form*

lxvh8x XT,RA,RB

31	T	RA	RB	812	TX
0	6	11	16	21	31

```
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 7
   VSR[32×TX+T].hword[i] ← MEM(EA+2×i, 2)
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 i from 0 to 7, do the following.

When Big-Endian byte ordering is employed, the contents of the halfword in storage at address EA+2×i are placed into load_data in such an order that;

- the contents of the byte in storage at address EA+2×i are placed into byte element 0 of load_data,
- the contents of the byte in storage at address EA+2×i+1 are placed into byte element 1 of load_data.

When Little-Endian byte ordering is employed, the contents of the quadword in storage at address EA are placed into VSR[XT] in such an order that;

- the contents of the byte in storage at address EA+2×i are placed into byte element 1 of load_data,
- the contents of the byte in storage at address EA+2×i+1 are placed into byte element 0 of load_data.

load_data is placed into halfword element i of VSR[XT].

Special Registers Altered:

None

Programming Note

lxvd2x, ***lxvw4x***, ***lxvh8x***, ***lxvb16x***, and ***lxvx*** exhibit identical behavior in Big-Endian mode.

Example: Loading data using *Load VSX Vector Halfword*8 Indexed*

```
short  X[] = { 0x0001, 0x1011, 0x2021, 0x3031,
               0x4041, 0x5051, 0x6061, 0x7071 };
```

Big-endian storage image of X

addr(X):	00	01	10	11	20	21	30	31	40	41	50	51	60	61	70	71
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

Little-endian storage image of X

addr(X):	01	00	11	10	21	20	31	30	41	40	51	50	61	60	71	70
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

Loading a vector of 8 halfword elements from Big-Endian storage in VSR[XT] using ***lxvh8x***, retaining left-to-right element ordering.

```
# Assumptions
#   GPR[PX] = address of X
lxvh8x   xX,r0,rPX
```

VSR[X]:	00	01	10	11	20	21	30	31	40	41	50	51	60	61	70	71
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

Loading a vector of 8 halfword elements from Little-Endian storage in VSR[XT] using ***lxvh8x***, retaining left-to-right element ordering.

```
# Assumptions
#   GPR[PX] = address of X
lxvh8x   xX,r0,rPX
```

VSR[X]:	00	01	10	11	20	21	30	31	40	41	50	51	60	61	70	71
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

VSR Data Layout for lxvh8x

tgt	VSR[XT].hword[0]	VSR[XT].hword[1]	VSR[XT].hword[2]	VSR[XT].hword[3]	VSR[XT].hword[4]	VSR[XT].hword[5]	VSR[XT].hword[6]	VSR[XT].hword[7]
	0	16	32	48	64	80	96	112 127

*Load VSX Vector Word*4 Indexed X-form*

lxvw4x XT,RA,RB

31	T	RA	RB	780	TX
0	6	11	16	21	31

```
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].

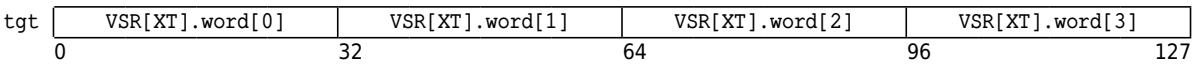
Special Registers Altered:

None

Programming Note

lxvd2x, ***lxvw4x***, ***lxvh8x***, ***lxvb16x***, and ***lxvx*** exhibit identical behavior in Big-Endian mode.

VSR Data Layout for lxvw4x



Load VSX Vector Indexed X-form

lxvx XT,RA,RB

31	T	RA	RB	4	/	12	TX
0	6	11	16	21	25	26	31

```
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] ← MEM(EA,16)
```

Let XT be the value 32×TX + T.

Let the effective address (EA) be the sum of the contents of GPR[RA], or 0 if RA is equal to 0, and the contents of GPR[RB].

When Big-Endian byte ordering is employed, the contents of the quadword in storage at address EA are placed into load_data in such an order that;

- the contents of the byte in storage at address EA are placed into byte element 0 of load_data,
- the contents of the byte in storage at address EA+1 are placed into byte element 1 of load_data, and so forth until
- the contents of the byte in storage at address EA+15 are placed into byte element 15 of load_data.

When Little-Endian byte ordering is employed, the contents of the quadword in storage at address EA are placed into load_data in such an order that;

- the contents of the byte in storage at address EA are placed into byte element 15 of load_data,
- the contents of the byte in storage at address EA+1 are placed into byte element 14 of load_data, and so forth until
- the contents of the byte in storage at address EA+15 are placed into byte element 0 of load_data.

load_data is placed into VSR[XT].

Special Registers Altered:

None

VSR Data Layout for lxvx

Example: Loading data using *Load 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, 0xF2F3, 0xF4F5, 0xF6F7, 0xE0E1, 0xE2E3, 0xE4E5, 0xE6E7 };
float   Y[4]  = { 0xF0F1_F2F3, 0xF4F5_F6F7, 0xE0E1_E2E3, 0xE4E5_E6E7 };
double  Z[2]  = { 0xF0F1_F2F3_F4F5_F6F7, 0xE0E1_E2E3_E4E5_E6E7 };
```

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)	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
+0x00:	F0	F1	F2	F3	F4	F5	F6	F7	E0	E1	E2	E3	E4	E5	E6	E7
+0x10:	F0	F1	F2	F3	F4	F5	F6	F7	E0	E1	E2	E3	E4	E5	E6	E7
+0x20:	F0	F1	F2	F3	F4	F5	F6	F7	E0	E1	E2	E3	E4	E5	E6	E7
+0x30:	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

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
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]:	E0	E1	E2	E3	E4	E5	E6	E7	F0	F1	F2	F3	F4	F5	F6	F7
VSR[Z]:	F0	F1	F2	F3	F4	F5	F6	F7	E0	E1	E2	E3	E4	E5	E6	E7

Loading 16 bytes of data from Little-Endian storage in `VSR[XT]` using ***lxvx***.

Little-endian storage image of W, X, Y, & Z

addr(W)	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
+0x00:	F0	F1	F2	F3	F4	F5	F6	F7	E0	E1	E2	E3	E4	E5	E6	E7
+0x10:	F1	F0	F3	F2	F5	F4	F7	F6	E1	E0	E3	E2	E5	E4	E7	E6
+0x20:	F3	F2	F1	F0	F7	F6	F5	F4	E3	E2	E1	E0	E7	E6	E5	E4
+0x30:	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

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
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

Load VSX Vector Doubleword & Splat Indexed X-form

lxvdsx XT,RA,RB

31	T	RA	RB	332	TX
0	6	11	16	21	31

```
if MSR.VSX=0 then VSX_Unavailable()

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;

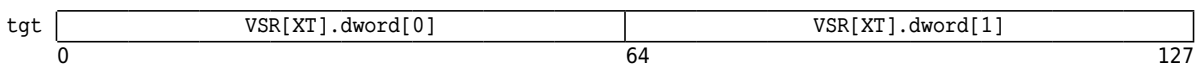
- the contents of the byte in storage at address EA are placed into byte element 7 of load_data,
- the contents of the byte in storage at address EA+1 are placed into byte element 6 of load_data, and so forth until
- the contents of the byte in storage at address EA+7 are placed into byte element 0 of load_data.

load_data is copied into each doubleword element of VSR[XT].

Special Registers Altered:

None

VSR Data Layout for lxvdsx



Load VSX Vector Word & Splat Indexed X-form

lxvwsx XT,RA,RB

31	T	RA	RB	364	TX
0	6	11	16	21	31

```
if TX=0 & MSR.VSX=0 then VSX_Unavailable()
if TX=1 & MSR.VEC=0 then Vector_Unavailable()

EA ← ((RA=0) ? 0 : GPR[RA]) + GPR[RB]

load_data ← MEM(EA,4)

do i = 0 to 3
   VSR[32×TX+T].word[i] ← load_data
end
```

Let XT be the value 32×TX + T.

Let the effective address (EA) be the sum of the contents of GPR[RA], or 0 if RA is equal to 0, and the contents of GPR[RB].

When Big-Endian byte ordering is employed, the contents of the word in storage at address EA are placed into load_data in such an order that;

- the contents of the byte in storage at address EA are placed into byte element 0 of load_data,
- the contents of the byte in storage at address EA+1 are placed into byte element 1 of load_data,
- the contents of the byte in storage at address EA+2 are placed into byte element 2 of load_data, and
- the contents of the byte in storage at address EA+3 are placed into byte element 3 of load_data.

When Little-Endian byte ordering is employed, the contents of the quadword in storage at address EA are placed into load_data in such an order that;

- the contents of the byte in storage at address EA are placed into byte element 3 of load_data,
- the contents of the byte in storage at address EA+1 are placed into byte element 2 of load_data,
- the contents of the byte in storage at address EA+2 are placed into byte element 1 of load_data, and
- the contents of the byte in storage at address EA+3 are placed into byte element 0 of load_data.

load_data is copied into each word element of VSR[XT].

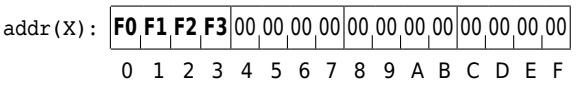
Special Registers Altered:

None

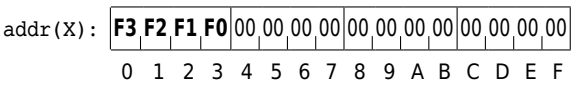
Example: Loading data using *Load VSX Vector Word & Splat Indexed*

```
int        X  = 0xF0F1_F2F3;
```

Big-endian storage image of X



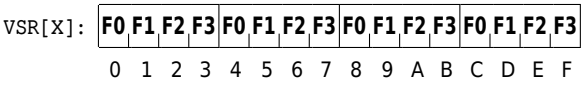
Little-endian storage image of X



Loading scalar word data from Big-Endian storage in VSR[XT] using ***lxvwsx***.

```
# Assumptions
#   GPR[PX] = address of X
lxvwsx   xX,r0,rPX
```

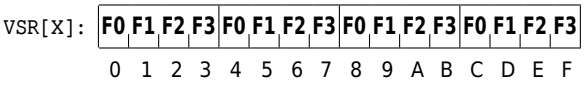
Final state of VSR X



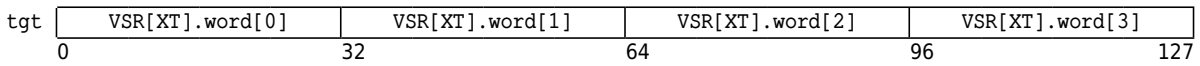
Loading scalar word data from Little-Endian storage in VSR[XT] using ***lxvwsx***.

```
# Assumptions
#   GPR[PX] = address of X
lxvwsx   xX,r0,rPX
```

Final state of VSR X



VSR Data Layout for lxvwsx



Load VSX Vector Rightmost Byte Indexed X-form

lxvrbx XT,RA,RB

31	T	RA	RB	13	TX
0	6	11	16	21	31

```
if MSR.VSX=0 then VSX_Unavailable()

EA = ((RA=0) ? 0 : GPR[RA]) + GPR[RB]
VSR[32×TX+T] = EXTZ128(MEM(EA,1))
```

Let XT be the value of 32×TX + T.

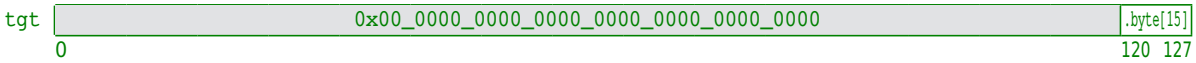
Let EA be the sum of GPR[RA], or 0 if RA=0, and GPR[RB].

Load the contents of the byte in storage at address EA into byte element 15 of VSR[XT]. The contents of byte elements 0-14 of VSR[XT] are set to 0.

Special Registers Altered:

None

VSR Data Layout for lxvrbx



Load VSX Vector Rightmost Doubleword Indexed X-form

lxvrdx XT,RA,RB

31	T	RA	RB	109	TX
0	6	11	16	21	31

```
if MSR.VSX=0 then VSX_Unavailable()

EA = ((RA=0) ? 0 : GPR[RA]) + GPR[RB]
VSR[32×TX+T] = EXTZ128(MEM(EA,8))
```

Let XT be the value of 32×TX + T.

Let EA be the sum of GPR[RA], or 0 if RA=0, and GPR[RB].

Load the contents of the doubleword in storage at address EA into doubleword element 1 of VSR[XT]. The contents of doubleword element 0 of VSR[XT] are set to 0.

When Big-Endian byte ordering is employed, the contents of the doubleword in storage at address EA are placed into load_data in such an order that;

- the contents of the byte in storage at address EA are placed into byte 0 of load_data,
- the contents of the byte in storage at address EA+1 are placed into byte 1 of load_data, and so forth until
- the contents of the byte in storage at address EA+7 are placed into byte 7 of load_data.

When Little-Endian byte ordering is employed, the contents of the doubleword in storage at address EA are placed into load_data in such an order that;

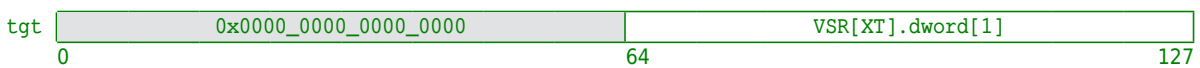
- the contents of the byte in storage at address EA are placed into byte 7 of load_data,
- the contents of the byte in storage at address EA+1 are placed into byte 6 of load_data, and so forth until
- the contents of the byte in storage at address EA+7 are placed into byte 0 of load_data.

load_data is placed into doubleword element 1 of VSR[XT]. The contents of doubleword element 0 of VSR[XT] are set to 0.

Special Registers Altered:

None

VSR Data Layout for lxvrdx



Load VSX Vector Rightmost Halfword Indexed X-form

lxvrhx XT,RA,RB

31	T	RA	RB	45	TX
0	6	11	16	21	31

```
if MSR.VSX=0 then VSX_Unavailable()

EA = ((RA=0) ? 0 : GPR[RA]) + GPR[RB]
VSR[32×TX+T] = EXTZ128(MEM(EA,2))
```

Let XT be the value of 32×TX + T.

Let EA be the sum of GPR[RA], or 0 if RA=0, and GPR[RB].

When Big-Endian byte ordering is employed, the contents of the halfword 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, and
- the contents of the byte in storage at address EA+1 are placed into byte 1 of load_data.

When Little-Endian byte ordering is employed, the contents of the halfword in storage at address EA are placed into load_data in such an order that;

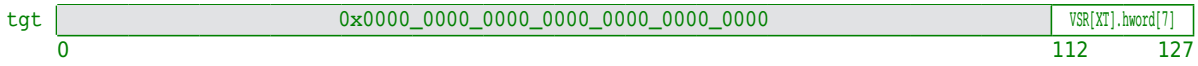
- the contents of the byte in storage at address EA are placed into byte 1 of load_data, and
- the contents of the byte in storage at address EA+1 are placed into byte 0 of load_data.

load_data is placed into halfword element 7 of VSR[XT]. The contents of halfword elements 0-6 of VSR[XT] are set to 0.

Special Registers Altered:

None

VSR Data Layout for lxvrhx



Load VSX Vector Rightmost Word Indexed X-form

lxvrwx XT,RA,RB

31	T	RA	RB	77	TX
0	6	11	16	21	31

```
if MSR.VSX=0 then VSX_Unavailable()

EA = ((RA=0) ? 0 : GPR[RA]) + GPR[RB]
VSR[32×TX+T] = EXTZ128(MEM(EA,4))
```

Let XT be the value of 32×TX + T.

Let EA be the sum of GPR[RA], or 0 if RA=0, and GPR[RB].

When Big-Endian byte ordering is employed, the contents of the word in storage at address EA are placed into load_data in such an order that;

- the contents of the byte in storage at address EA are placed into byte 0 of load_data,
- the contents of the byte in storage at address EA+1 are placed into byte 1 of load_data,
- the contents of the byte in storage at address EA+2 are placed into byte 2 of load_data,
- the contents of the byte in storage at address EA+3 are placed into byte 3 of load_data, and

When Little-Endian byte ordering is employed, the contents of the word in storage at address EA are placed into load_data in such an order that;

- the contents of the byte in storage at address EA are placed into byte 3 of load_data,
- the contents of the byte in storage at address EA+1 are placed into byte 2 of load_data,
- the contents of the byte in storage at address EA+2 are placed into byte 1 of load_data, and
- the contents of the byte in storage at address EA+3 are placed into byte 0 of load_data.

load_data is placed into word element 3 of VSR[XT]. The contents of word elements 0-2 of VSR[XT] are set to 0.

Special Registers Altered:

None

VSR Data Layout for lxvrwx

Load VSX Vector with Length X-form

lxvl XT,RA,RB

31	T	RA	RB	269	TX
0	6	11	16	21	31

```
if TX=0 & MSR.VSX=0 then VSX_Unavailable()
if TX=1 & MSR.VEC=0 then Vector_Unavailable()

EA ← (RA=0) ? 0 : GPR[RA]
nb ← EXTZ(GPR[RB].bit[0:7])
if nb>16 then nb ← 16

load_data ← 0x0000_0000_0000_0000_0000_0000_0000_0000

if MSR.LE = 0 then  // Big-Endian byte-ordering
   load_data.byte[0:nb-1]   ← MEM(EA,nb)
else               // Little-Endian byte-ordering
   load_data.byte[16-nb:15] ← MEM(EA,nb)

VSR[32×TX+T] ← load_data
```

Let XT be the value 32×TX + T.

Let the effective address (EA) be the contents of GPR[RA], or 0 if RA is equal to 0.

Let nb be the unsigned integer value in bits 0:7 of GPR[RB].

If nb is equal to 0, the storage access is not performed and the contents of VSR[XT] are set to 0.

Otherwise, 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:

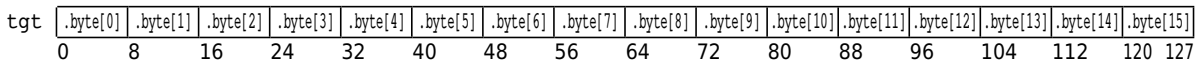
None

Programming Note

Loading less than 16 bytes of data using ***lxvl*** in BE mode results in data being loaded into the target VSR left-to-right, placing the first byte in the left-most byte of the target VSR, and padded on the right with 0s.

Loading less than 16 bytes of data using ***lxvl*** in LE mode results in data being loaded into the target VSR right-to-left, placing the first byte in the right-most byte of the target VSR, and padded on the left with 0s.

VSR Data Layout for lxvl



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

addr(S)	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
+0x00:	'T'	'h'	'i'	's'	' '	'i'	's'	' '	'a'	' '	'T'	'E'	'S'	'T'	E0	E1
+0x10:	E2	E3	E4	E5	E6	E7	E8	E9	EA	EB	F0	F1	F2	F3	F4	F5
+0x20:	F6	F7	F8	F9	00	00	00	00	00	00	00	00	00	00	00	00

```
# 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
```

VSR register image of S, X, & Z

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
VSR[S]:	'T'	'h'	'i'	's'	' '	'i'	's'	' '	'a'	' '	'T'	'E'	'S'	'T'	00	00
VSR[X]:	E0	E1	E2	E3	E4	E5	E6	E7	E8	E9	EA	EB	00	00	00	00
VSR[Z]:	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	00	00	00	00	00	00

Loading less than 16-byte data from Little-Endian storage in `VSR[XT]` using ***lxvl***.

Little-endian storage image of S, X, & Z

addr(S)	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
+0x00:	'T'	'h'	'i'	's'	' '	'i'	's'	' '	'a'	' '	'T'	'E'	'S'	'T'	E1	E0
+0x10:	E3	E2	E5	E4	E7	E6	E9	E8	EB	EA	F9	F8	F7	F6	F5	F4
+0x20:	F3	F2	F1	F0	00	00	00	00	00	00	00	00	00	00	00	00

```
# 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
```

VSR register image of S, X, & Z

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
VSR[S]:	00	00	'T'	'S'	'E'	'T'	' '	'a'	' '	's'	'i'	' '	's'	'i'	'h'	'T'
VSR[X]:	00	00	00	00	EA	EB	E8	E9	E6	E7	E4	E5	E2	E3	E0	E1
VSR[Z]:	00	00	00	00	00	00	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9

Load VSX Vector with Length Left-justified X-form

lxvll XT,RA,RB

31	T	RA	RB	301	TX
0	6	11	16	21	31

```
if TX=0 & MSR.VSX=0 then VSX_Unavailable()
if TX=1 & MSR.VEC=0 then Vector_Unavailable()

EA ← (RA=0) ? 0 : GPR[RA]
nb ← EXTZ(GPR[RB].bit[0:7])
if nb > 16 then nb ← 16
if nb > 0 then do i = 0 to nb-1
   VSR[32×TX+T].byte[i] ← MEM(EA+i,1)
end
if nb < 16 then do i = nb to 15
   VSR[32×TX+T].byte[i] ← 0x00
end
```

Let XT be the value 32×TX + T.

Let the effective address (EA) be the contents of GPR[RA], or 0 if RA is equal to 0.

Let nb be the unsigned integer value in bits 0:7 of GPR[RB].

If nb is equal to 0, the storage access is not performed and the contents of VSR[XT] are set to 0.

Otherwise, do the following.

If nb less than 16, the contents of the nb bytes in storage starting at address EA are placed into the leftmost nb bytes of VSR[XT], and the contents of the rightmost 16-nb bytes of VSR[XT] are set to 0x00.

Otherwise, the contents of the quadword in storage at address EA are placed into VSR[XT].

Data is loaded from storage into VSR[XT] in Big-Endian byte ordering (i.e., the byte in storage at address EA is placed into byte element 0 of VSR[XT], the byte in storage at address EA+1 is placed in byte element 1 of VSR[XT], and so forth).

If the contents of bits 8:63 of GPR[RB] are not equal to 0, the results are boundedly undefined.

Special Registers Altered:

None

Example: Loading less than 16-byte left-justified data

```
decimal  X = +1234567890123456789;
decimal  Y = -123456;
decimal  Z = +1004966723510220;
```

Loading less than 16-byte data from storage in VSR[XT], left-justified, using ***lxvll***.

Initial state of VSRs X, Y, & Z

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
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

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
X+0x00:	**12**	**34**	**56**	**78**	**90**	**12**	**34**	**56**	**78**	**9C**	**01**	**23**	**45**	**6D**	**01**	**00**
X+0x10:	**49**	**66**	**72**	**35**	**10**	**22**	**0C**	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

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
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

VSR Data Layout for lxvll

tgt	.byte[0]	.byte[1]	.byte[2]	.byte[3]	.byte[4]	.byte[5]	.byte[6]	.byte[7]	.byte[8]	.byte[9]	.byte[10]	.byte[11]	.byte[12]	.byte[13]	.byte[14]	.byte[15]
	0	8	16	24	32	40	48	56	64	72	80	88	96	104	112	120 127

Programming Note

lxvll always performs storage accesses using Big-Endian byte-ordering. As such, care must be taken when using these instructions in Little-Endian systems.

Store VSX Vector DQ-form

stxv XS,disp(RA)

61	S	RA	DQ	SX	5
0	6	11	16	28	29 31

Prefixed Store VSX Vector 8LS:D-form

pstxv XS,D(RA),R

Prefix:

1	0	0	//	R	//	d0
0	6	8	9	11	12	14 31

Suffix:

27	SX	S	RA	d1
0	5	6	11	16 31

```
if ''stxv'' & SX=0 & MSR.VSX=0 then VSX_Unavailable()
if ''stxv'' & SX=1 & MSR.VEC=0 then Vector_Unavailable()
if ''pstxv'' & MSR.VSX=0 then VSX_Unavailable()

if ''stxv''  then
   EA ← (RA|0) + EXTS64(DQ||0b0000)
if ''pstxv'' & R=0 then
   EA ← (RA|0) + EXTS64(d0||d1)
if ''pstxv'' & R=1 then
   EA ← CIA + EXTS64(d0||d1)

MEM(EA,16) ← VSR[32×SX+S]
```

Let XS be the value 32×SX + S.

For ***stxv***, let the effective address (EA) be the sum of the contents of GPR[RA], or the value 0 if RA=0, and the value DQ||0b0000, sign-extended to 64 bits.

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

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

Let store_data be the contents of VSR[XS].

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.

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

Special Registers Altered:

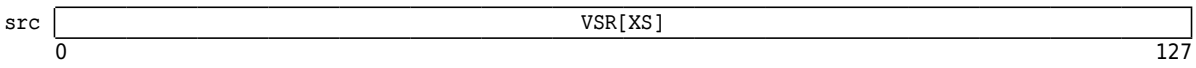
None

Extended Mnemonics:

Extended Mnemonics for *Prefixed Store VSX Vector*:

Extended mnemonic:		Equivalent to:	
pstxv	XS,D(RA)	pstxv	XS,D(RA),0
pstxv	XS,D	pstxv	XS,D(0),1

VSR Data Layout for [p]stxv



*Store VSX Vector Byte*16 Indexed X-form*

stxvb16x XS,RA,RB

31	S	RA	RB	1004	SX
0	6	11	16	21	31

```
if SX=0 & MSR.VSX=0 then VSX_Unavailable()
if SX=1 & MSR.VEC=0 then Vector_Unavailable()

EA ← ((RA=0) ? 0 : GPR[RA]) + GPR[RB]

do i = 0 to 15
   MEM(EA+i,1) ← VSR[32×SX+S].byte[i]
end
```

Let XS be the value 32×SX + S.

Let the effective address (EA) be the sum of the contents of GPR[RA], or 0 if RA is equal to 0, and the contents of GPR[RB].

For each integer value from 0 to 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:

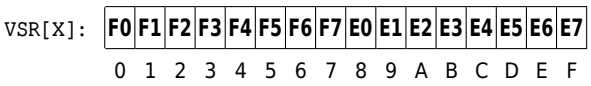
None

Programming Note

stxvd2x, ***stxvw4x***, ***stxvh8x***, ***stxvb16x***, and ***stxvx*** exhibit identical behavior in Big-Endian mode.

Example: Storing data using *Store VSX Vector Byte*16 Indexed*

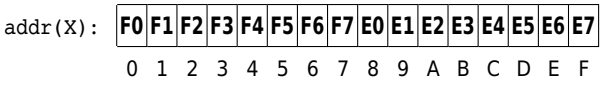
```
char        X[16];
```



Storing a vector of 16 byte elements from VSR[XS] into Big-Endian storage using ***stxvb16x***, retaining left-to-right element ordering.

```
# Assumptions
#   GPR[PX] = address of X

stxvb16x   xX,r0,rPX
```

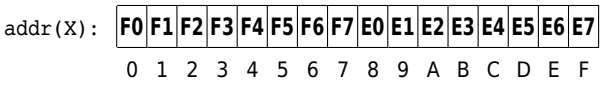
Big-endian storage image of X



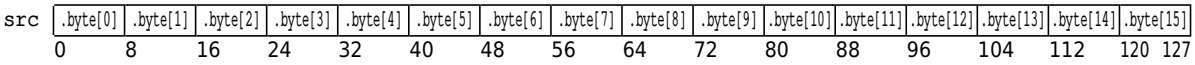
Storing a vector of 16 byte elements from VSR[XS] into Little-Endian storage using ***stxvb16x***, retaining left-to-right element ordering.

```
# Assumptions
#   GPR[PX] = address of X

stxvb16x   xX,r0,rPX
```

Little-endian storage image of X



VSR Data Layout for stxvb16x



*Store VSX Vector Doubleword*2 Indexed X-form*

stxvd2x XS,RA,RB

31	S	RA	RB	972	SX
0	6	11	16	21	31

```
if MSR.VSX=0 then VSX_Unavailable()

EA ← ((RA=0) ? 0 : GPR[RA]) + GPR[RB]

MEM(EA,8)     ← VSR[32×SX+S].dword[0]
MEM(EA+8,8)   ← VSR[32×SX+S].dword[1]
```

Let XS be the value 32×SX + S.

Let EA be the sum of the contents of GPR[RA], or 0 if RA is equal to 0, and the contents of GPR[RB].

For each integer value i from 0 to 1, do the following.

Let store_data be the contents of doubleword element i of VSR[XS].

When Big-Endian byte ordering is employed, store_data is placed in the doubleword in storage at address EA+8×i in such order that;

- byte 0 of store_data is placed into the byte in storage at address EA+8×i,
- byte 1 of store_data is placed into the byte in storage at address EA+8×i+1, and so forth until
- byte 7 of store_data is placed into the byte in storage at address EA+8×i+7.

When Little-Endian byte ordering is employed, store_data is placed in the doubleword in storage at address EA+8×i in such order that;

- byte 0 of store_data is placed into the byte in storage at address EA+8×i+7,
- byte 1 of store_data is placed into the byte in storage at address EA+8×i+6, and so forth until
- byte 7 of store_data is placed into the byte in storage at address EA+8×i.

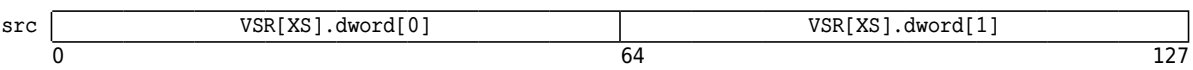
Special Registers Altered:

None

Programming Note

stxvd2x, ***stxvw4x***, ***stxvh8x***, ***stxvb16x***, and ***stxvx*** exhibit identical behavior in Big-Endian mode.

VSR Data Layout for stxvd2x



*Store VSX Vector Halfword*8 Indexed X-form*

stxvh8x XS,RA,RB

31	S	RA	RB	940	SX
0	6	11	16	21	31

```
if SX=0 & MSR.VSX=0 then VSX_Unavailable()
if SX=1 & MSR.VEC=0 then Vector_Unavailable()

EA ← ((RA=0) ? 0 : GPR[RA]) + GPR[RB]

do i = 0 to 7
   MEM(EA+2×i,2) ← VSR[32×SX+S].hword[i]
end
```

Let XS be the value 32×SX + S.

Let the effective address (EA) be the sum of the contents of GPR[RA], or 0 if RA is equal to 0, and the contents of GPR[RB].

For each integer value from 0 to 7, do the following.

The contents of byte element i of VSR[XS] are placed into the byte in storage at address EA+i.

For each integer value from 0 to 7, do the following.

When Big-Endian byte ordering is employed, the contents of halfword element i of VSR[XS] are placed into the halfword in storage at address EA+2×i in such an order that;

- the contents of byte sub-element 0 of halfword element i of VSR[XS] are placed into the byte in storage at address EA+2×i, and
- the contents of byte sub-element 1 of halfword element i of VSR[XS] are placed into the byte in storage at address EA+2×i+1.

When Little-Endian byte ordering is employed, the contents of halfword element i of VSR[XS] are placed into the halfword in storage at address EA+2×i in such an order that;

- the contents of byte sub-element 1 of halfword element i of VSR[XS] are placed into the byte in storage at address EA+2×i, and
- the contents of byte sub-element 0 of halfword element i of VSR[XS] are placed into the byte in storage at address EA+2×i+1.

Special Registers Altered:

None

Example: Storing data using *Store VSX Vector Halfword*8 Indexed*

```
short     X[8];
```

VSR[X]:

00	01	10	11	20	21	30	31	40	41	50	51	60	61	70	71
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

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
```

Big-endian storage image of X

addr(X):

00	01	10	11	20	21	30	31	40	41	50	51	60	61	70	71
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

Storing a vector of 8 halfword elements from VSR[X] into Little-Endian storage using ***stxvh8x***, retaining left-to-right element ordering.

```
# Assumptions
#   GPR[PX] = address of X

stxvh8x   xX,r0,rPX
```

Little-endian storage image of X

addr(X):

01	00	11	10	21	20	31	30	41	40	51	50	61	60	71	70
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

Programming Note

stxvd2x, ***stxvw4x***, ***stxvh8x***, ***stxvb16x***, and ***stxvx*** exhibit identical behavior in Big-Endian mode.

VSR Data Layout for stxvh8x

src	VSR[XS].hword[0]	VSR[XS].hword[1]	VSR[XS].hword[2]	VSR[XS].hword[3]	VSR[XS].hword[4]	VSR[XS].hword[5]	VSR[XS].hword[6]	VSR[XS].hword[7]
	0	16	32	48	64	80	96	112 127

*Store VSX Vector Word*4 Indexed X-form*

stxvw4x XS,RA,RB

31	S	RA	RB	908	SX
0	6	11	16	21	31

```
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×SX + S.

Let EA be the sum of the contents of GPR[RA], or 0 if RA is equal to 0, and the contents of GPR[RB].

For each integer value i from 0 to 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+4×i in such order that;

- byte 0 of store_data is placed into the byte in storage at address EA+4×i,
- byte 1 of store_data is placed into the byte in storage at address EA+4×i+1, and so forth until
- byte 3 of store_data is placed into the byte in storage at address EA+4×i+3.

When Little-Endian byte ordering is employed, store_data is placed in the word in storage at address EA+4×i in such order that;

- byte 0 of store_data is placed into the byte in storage at address EA+4×i+3,
- byte 1 of store_data is placed into the byte in storage at address EA+4×i+2, and so forth until
- byte 3 of store_data is placed into the byte in storage at address EA+4×i.

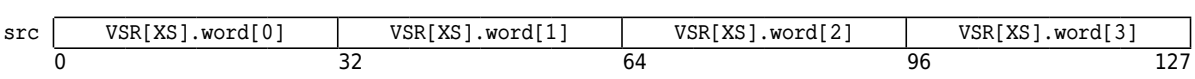
Special Registers Altered:

None

Programming Note

stxvd2x, ***stxvw4x***, ***stxvh8x***, ***stxvb16x***, and ***stxvx*** exhibit identical behavior in Big-Endian mode.

VSR Data Layout for stxvw4x



Store VSX Vector Indexed X-form

stxvx XS,RA,RB

31	S	RA	RB	396	SX
0	6	11	16	21	31

```
if SX=0 & MSR.VSX=0 then VSX_Unavailable()
if SX=1 & MSR.VEC=0 then Vector_Unavailable()

EA ← ((RA=0) ? 0 : GPR[RA]) + GPR[RB]

MEM(EA,16) ← VSR[32×SX+S]
```

Let XS be the value 32×SX + S.

Let the effective address (EA) be the sum of the contents of GPR[RA], or 0 if RA is equal to 0, and the contents of GPR[RB].

When Big-Endian byte ordering is employed, store_data is placed into the quadword in storage at address EA in such an order that;

- byte 0 of store_data is placed into the byte in storage at address EA,
- byte 1 of store_data is placed into the byte in storage at address EA+1, and so forth until
- byte 15 of store_data is placed into the byte in storage at address EA+15.

When Little-Endian byte ordering is employed, store_data is placed into the quadword in storage at address EA in such an order that;

- byte 15 of store_data is placed into the byte in storage at address EA,
- byte 14 of store_data is placed into the byte in storage at address EA+1, and so forth until
- byte 0 of store_data is placed into the byte in storage at address EA+15.

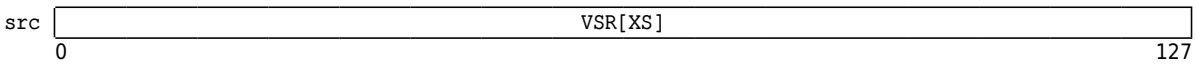
Special Registers Altered:

None

Programming Note

stxvd2x, ***stxvw4x***, ***stxvh8x***, ***stxvb16x***, and ***stxvx*** exhibit identical behavior in Big-Endian mode.

VSR Data Layout for stxvx



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, 0xF2F3, 0xF4F5, 0xF6F7, 0xE0E1, 0xE2E3, 0xE4E5, 0xE6E7 };
float   Y[4]  = { 0xF0F1_F2F3, 0xF4F5_F6F7, 0xE0E1_E2E3, 0xE4E5_E6E7 };
double  Z[2]  = { 0xF0F1_F2F3_F4F5_F6F7, 0xE0E1_E2E3_E4E5_E6E7 };
```

Storing 16 bytes of data into Big-Endian storage from `VSR[XS]` using ***stxvx***.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
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

```
# 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

stxvx   xW,r0,rPW
stxvx   xX,r0,rPX
stxvx   xY,r0,rPY
stxvx   xZ,r0,rPZ
```

addr(W) Big-endian storage image of W, X, Y, & Z

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
+0x00:	F0	F1	F2	F3	F4	F5	F6	F7	E0	E1	E2	E3	E4	E5	E6	E7
+0x10:	F0	F1	F2	F3	F4	F5	F6	F7	E0	E1	E2	E3	E4	E5	E6	E7
+0x20:	F0	F1	F2	F3	F4	F5	F6	F7	E0	E1	E2	E3	E4	E5	E6	E7
+0x30:	F0	F1	F2	F3	F4	F5	F6	F7	E0	E1	E2	E3	E4	E5	E6	E7

Storing 16 bytes of data into Little-Endian storage from `VSR[XS]` using ***stxvx***.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
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

```
# 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

stxvx   xW,r0,rPW
stxvx   xX,r0,rPX
stxvx   xY,r0,rPY
stxvx   xZ,r0,rPZ
```

addr(W) Little-endian storage image of W, X, Y, & Z

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
+0x00:	F0	F1	F2	F3	F4	F5	F6	F7	E0	E1	E2	E3	E4	E5	E6	E7
+0x10:	F1	F0	F3	F2	F5	F4	F7	F6	E1	E0	E3	E2	E5	E4	E7	E6
+0x20:	F3	F2	F1	F0	F7	F6	F5	F4	E3	E2	E1	E0	E7	E6	E5	E4
+0x30:	F7	F6	F5	F4	F3	F2	F1	F0	E7	E6	E5	E4	E3	E2	E1	E0

Store VSX Vector Rightmost Byte Indexed X-form

stxvrbx XS,RA,RB

31	S	RA	RB	141	SX
0	6	11	16	21	31

```
if MSR.VSX=0 then VSX_Unavailable()

EA = ((RA=0) ? 0 : GPR[RA]) + GPR[RB];
MEM(EA,1) = VSR[32×SX+S].byte[15];
```

Let XS be the value of 32×SX + S.

Let EA be the sum of GPR[RA], or 0 if RA=0, and GPR[RB].

The contents of byte element 15 of VSR[XS] are placed into storage at address EA.

Special Registers Altered:

None

Store VSX Vector Rightmost Doubleword Indexed X-form

stxvrdx XS,RA,RB

31	S	RA	RB	237	SX
0	6	11	16	21	31

```
if MSR.VSX=0 then VSX_Unavailable()

EA = ((RA=0) ? 0 : GPR[RA]) + GPR[RB]
MEM(EA,8) = VSR[32×SX+S].dword[1]
```

Let XS be the value of 32×SX + S.

Let EA be the sum of GPR[RA], or 0 if RA=0, and GPR[RB].

Let store_data be the contents of doubleword element 1 of VSR[XS].

When Big-Endian byte ordering is employed, store_data is placed into the doubleword in storage at address EA in such an order that;

- byte 0 of store_data is placed into the byte in storage at address EA,
- byte 1 of store_data is placed into the byte in storage at address EA+1, and so forth until
- byte 7 of store_data is placed into the byte in storage at address EA+7.

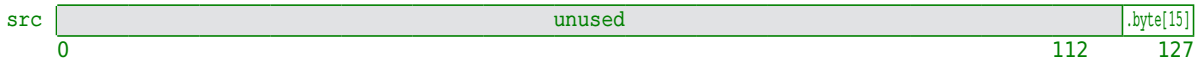
When Little-Endian byte ordering is employed, store_data is placed into the doubleword in storage at address EA in such an order that;

- byte 7 of store_data is placed into the byte in storage at address EA,
- byte 6 of store_data is placed into the byte in storage at address EA+1, and so forth until
- byte 0 of store_data is placed into the byte in storage at address EA+7.

Special Registers Altered:

None

VSR Data Layout for stxvrbx



VSR Data Layout for stxvrdx

Store VSX Vector Rightmost Halfword Indexed X-form

stxvrhx XS,RA,RB

31	S	RA	RB	173	SX
0	6	11	16	21	31

```
if MSR.VSX=0 then VSX_Unavailable()

EA = ((RA=0) ? 0 : GPR[RA]) + GPR[RB]
MEM(EA,2) = VSR[32×SX+S].hword[7]
```

Let XS be the value of 32×SX + S.

Let EA be the sum of GPR[RA], or 0 if RA=0, and GPR[RB].

Let store_data be the contents of halfword element 7 of VSR[XS].

When Big-Endian byte ordering is employed, store_data is placed into the halfword in storage at address EA in such an order that;

- byte 0 of store_data is placed into the byte in storage at address EA,
- byte 1 of store_data is placed into the byte in storage at address EA+1.

When Little-Endian byte ordering is employed, store_data is placed into the halfword in storage at address EA in such an order that;

- byte 1 of store_data is placed into the byte in storage at address EA,
- byte 0 of store_data is placed into the byte in storage at address EA+1.

Special Registers Altered:

None

Store VSX Vector Rightmost Word Indexed X-form

stxvrwx XS,RA,RB

31	S	RA	RB	205	SX
0	6	11	16	21	31

```
if MSR.VSX=0 then VSX_Unavailable()

EA = ((RA=0) ? 0 : GPR[RA]) + GPR[RB]
MEM(EA,4) = VSR[32×SX+S].word[3]
```

Let XS be the value of 32×SX + S.

Let EA be the sum of GPR[RA], or 0 if RA=0, and GPR[RB].

Let store_data be the contents of word element 3 of VSR[XS].

When Big-Endian byte ordering is employed, store_data is placed into the word in storage at address EA in such an order that;

- byte 0 of store_data is placed into the byte in storage at address EA,
- byte 1 of store_data is placed into the byte in storage at address EA+1, and so forth until
- byte 3 of store_data is placed into the byte in storage at address EA+3.

When Little-Endian byte ordering is employed, store_data is placed into the word in storage at address EA in such an order that;

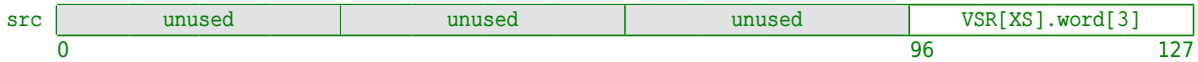
- byte 3 of store_data is placed into the byte in storage at address EA,
- byte 2 of store_data is placed into the byte in storage at address EA+1, and so forth until
- byte 0 of store_data is placed into the byte in storage at address EA+3.

Special Registers Altered:

None

VSR Data Layout for stxvrhx

VSR Data Layout for stxvrwx



Store VSX Vector with Length X-form

stxvl XS,RA,RB

31	S	RA	RB	397	SX
0	6	11	16	21	31

```
if SX=0 & MSR.VSX=0 then VSX_Unavailable()
if SX=1 & MSR.VEC=0 then Vector_Unavailable()

EA ← (RA=0) ? 0 : GPR[RA]

nb ← EXTZ(GPR[RB].bit[0:7])
if nb>16 then nb ← 16

if MSR.LE = 0 then       // Big-Endian byte-ordering
   store_data ← VSR[32×SX+S].byte[0:nb-1]
else                     // Little-Endian byte ordering
   store_data ← VSR[32×SX+S].byte[16-nb:15]

MEM(EA,nb) ← store_data
```

Let XS be the value 32×SX + S.

Let the effective address (EA) be the contents of GPR[RA], or 0 if RA is equal to 0.

Let nb be the unsigned integer value in bits 0:7 of GPR[RB].

If nb is equal to 0, the storage access is not performed.

Otherwise, when Big-Endian byte-ordering is employed, do the following.

If nb less than 16, the contents of the leftmost nb bytes of VSR[XS] are placed in storage starting at address EA.

Otherwise, the contents of VSR[XS] are placed into the quadword in storage at address EA.

Otherwise, when Little-Endian byte ordering is employed, do the following.

If nb less than 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.

Special Registers Altered:

None

VSR Data Layout for stxvl

	.byte[0]	.byte[1]	.byte[2]	.byte[3]	.byte[4]	.byte[5]	.byte[6]	.byte[7]	.byte[8]	.byte[9]	.byte[10]	.byte[11]	.byte[12]	.byte[13]	.byte[14]	.byte[15]
src	0	8	16	24	32	40	48	56	64	72	80	88	96	104	112	120 127

Programming Note

Storing N bytes of data from the source VSR using ***stxvl*** in BE mode, results in the leftmost N bytes in the source VSR being placed in storage, starting with the leftmost byte of the source VSR.

Storing N bytes of data from the source VSR using ***stxvl*** in LE mode, results in the rightmost N bytes in the source VSR being placed in storage, starting with the rightmost byte of the source VSR.

Example: Storing less than 16-byte data from VSR using *stxvl*

```
char        S[14] = "This is a TEST";
short       X[6]  = { 0xE0E1, 0xE2E3, 0xE4E5, 0xE6E7, 0xE8E9, 0xEAEB };
binary80    Z     = 0xF0F1F2F3F4F5F6F7F8F9;
```

Storing less than 16-byte data in VSR[XS] into Big-Endian storage using ***stxvl***.

```
# Assumptions
#   GPR[NS] = 14 (length of S in # of bytes)
#   GPR[NX] = 12 (length of X in # of bytes)
#   GPR[NZ] = 10 (length of Z in # of bytes)
#   GPR[PS] = address of S
```

VSR register image of S, X, & Z

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
VSR[S]:	'T'	'h'	'i'	's'	' '	'i'	's'	' '	'a'	' '	'T'	'E'	'S'	'T'	00	00
VSR[X]:	E0	E1	E2	E3	E4	E5	E6	E7	E8	E9	EA	EB	00	00	00	00
VSR[Z]:	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	00	00	00	00	00	00

```
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
```

addr(S) Final state of Big-Endian storage image of S, X, & Z

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
+0x00:	'T'	'h'	'i'	's'	' '	'i'	's'	' '	'a'	' '	'T'	'E'	'S'	'T'	E0	E1
+0x10:	E2	E3	E4	E5	E6	E7	E8	E9	EA	EB	F0	F1	F2	F3	F4	F5
+0x20:	F6	F7	F8	F9	00	00	00	00	00	00	00	00	00	00	00	00

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

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
VSR[S]:	00	00	'T'	'S'	'E'	'T'	' '	'a'	' '	's'	'i'	' '	's'	'i'	'h'	'T'
VSR[X]:	00	00	00	00	EA	EB	E8	E9	E6	E7	E4	E5	E2	E3	E0	E1
VSR[Z]:	00	00	00	00	00	00	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9

```
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
```

addr(S) Final state of Little-Endian storage image of S, X, & Z

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
+0x00:	'T'	'h'	'i'	's'	' '	'i'	's'	' '	'a'	' '	'T'	'E'	'S'	'T'	E1	E0
+0x10:	E3	E2	E5	E4	E7	E6	E9	E8	EB	EA	F9	F8	F7	F6	F5	F4
+0x20:	F3	F2	F1	F0	00	00	00	00	00	00	00	00	00	00	00	00

Store VSX Vector with Length Left-justified X-form

stxvll XS,RA,RB

31	S	RA	RB	429	SX
0	6	11	16	21	31

```
if SX=0 & MSR.VSX=0 then VSX_Unavailable()
if SX=1 & MSR.VEC=0 then Vector_Unavailable()

EA ← (RA=0) ? 0 : GPR[RA]

nb ← EXTZ(GPR[RB].bit[0:7])
if nb>16 then nb ← 16

if nb>0 then do i = 0 to nb-1
   MEM(EA+i,1) ← VSR[32×SX+S].byte[i]
end
```

Let XS be the value 32×SX + S.

Let the effective address (EA) be the contents of GPR[RA], or 0 if RA is equal to 0.

Let nb be the unsigned integer value in bits 0:7 of GPR[RB].

If nb is equal to 0, the storage access is not performed.

Otherwise, do the following.

If nb less than 16, the contents of the leftmost nb bytes of VSR[XS] are placed in storage starting at address EA.

Otherwise, the contents of VSR[XS] are placed into the quadword in storage at address EA.

Data is stored from VSR[XS] into storage in Big-Endian byte ordering (i.e., the contents of byte element 0 of VSR[XS] are placed into the byte in storage at address EA, the contents of byte element 1 of VSR[XS] are placed into the byte in storage at address EA+1, and so forth).

If the contents of bits 8:63 of GPR[RB] are not equal to 0, the results are boundedly undefined.

Special Registers Altered:

None

Programming Note

stxvll always performs storage accesses using Big-Endian byte-ordering. As such, care must be taken when using these instructions in Little-Endian systems.

Example: Storing less than 16-byte left-justified data

```
decimal  X = +1234567890123456789;
decimal  Y = -123456;
decimal  Z = +1004966723510220;
```

Storing less than 16-byte data, left-justified in VSR[XS], into storage using ***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

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
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

Initial state of Big-Endian & Little-Endian storage image of X, Y, & Z

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
X+0x00:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
X+0x10:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

```
stxvll  xX,rPX,rNX
stxvll  xY,rPY,rNY
stxvll  xZ,rPZ,rNZ
```

Final state of Big-Endian & Little-Endian storage image of X, Y, & Z

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
X+0x00:	**01**	**34**	**67**	**78**	**90**	**12**	**34**	**56**	**78**	**9C**	**01**	**23**	**45**	**6D**	**01**	**00**
X+0x10:	**49**	**66**	**72**	**35**	**10**	**22**	**0C**	00	00	00	00	00	00	00	00	00

VSR Data Layout for stxvll

src	.byte[0]	.byte[1]	.byte[2]	.byte[3]	.byte[4]	.byte[5]	.byte[6]	.byte[7]	.byte[8]	.byte[9]	.byte[10]	.byte[11]	.byte[12]	.byte[13]	.byte[14]	.byte[15]
	0	8	16	24	32	40	48	56	64	72	80	88	96	104	112	120 127

7.6.2.1.3 VSX Vector Paired Storage Access Instructions

Load VSX Vector Paired DQ-form

lxvp XTp,disp(RA)

6	Tp	TX	RA	DQ	0
0	6	10	11	16	28 31

Prefixed Load VSX Vector Paired 8LS:D-form

plxvp XTp,D(RA),R

Prefix:

1	0	0	//	R	//	d0
0	6	8	9	11	12 14	31

Suffix:

58	Tp	TX	RA	d1
0	6	10	11	16 31

```
if MSR.VSX=0 then VSX_Unavailable()

EAbase ← (RA=0) ? 0 : GPR[RA]
if ''lxvp'' then
   EAdisp ← EXTS64(DQ || 0b0000)
if ''plxvp'' then

   EAdisp ← EXTS64(d0 || d1)
if ''lxvp''        then EA ← EAbase + EAdisp
if ''plxvp'' & R=0 then EA ← EAbase + EAdisp
if ''plxvp'' & R=1 then EA ← CIA    + EAdisp

load_data ← MEM(EA, 32)

VSR[32×TX+2×Tp]   ← load_data.bit[  0:127]
VSR[32×TX+2×Tp+1] ← load_data.bit[128:255]
```

Let XTp be the value 32×TX + 2×Tp (i.e., only even values of XTp can be encoded in the instruction).

Let EAbase be the contents of GPR[RA], or 0 if RA=0.

For ***lxvp***, let the effective address (EA) be the sum of the integer value in GPR[RA], or 0 if RA=0, and the value DQ||0b0000, sign-extended to 64 bits.

For *plxvp*, if R=0, let the effective address (EA) be the sum of the integer value in GPR[RA], or 0 if RA=0, and the value d0||d1, sign-extended to 64 bits.

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

When Big-Endian byte ordering is employed, the contents of the octword in storage at address EA are placed into load_data in such an order that;

- the contents of the byte in storage at address EA are placed into byte 0 of load_data,
- the contents of the byte in storage at address EA+1 are placed into byte 1 of load_data, and so forth until
- the contents of the byte in storage at address EA+31 are placed into byte 31 of load_data.

When Little-Endian byte ordering is employed, the contents of the octword in storage at address EA are placed into load_data in such an order that;

- the contents of the byte in storage at address EA are placed into byte 31 of load_data,
- the contents of the byte in storage at address EA+1 are placed into byte 30 of load_data, and so forth until
- the contents of the byte in storage at address EA+31 are placed into byte 0 of load_data.

Bits 0-127 of load_data are placed into VSR[XTp].

Bits 128-255 of load_data is placed into VSR[XTp+1].

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

Special Registers Altered:

None

Extended Mnemonics:

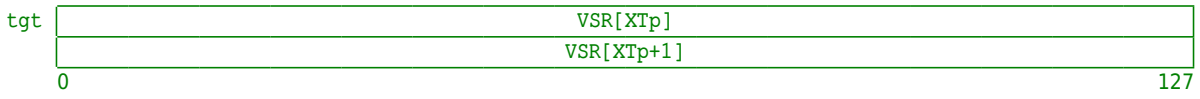
Extended Mnemonics for *Load VSX Vector Paired Prefixed*:

Extended mnemonic:		Equivalent to:	
plxvp	XTp,D(RA)	plxvp	XTp,D(RA),0
plxvp	XTp,D	plxvp	XTp,D(0),1

Programming Note

For best performance, EA should be word-aligned.

VSR Data Layout for [p]lxvp



Load VSX Vector Paired Indexed X-form

lxvpx XTp,RA,RB

31	Tp	TX	RA	RB	333	/
0	6	10	11	16	21	31

```
if MSR.VSX=0 then VSX_Unavailable()

EA ← ((RA=0) ? 0 : GPR[RA]) + GPR[RB]

load_data ← MEM(EA,32)
VSR[32×TX+2×Tp]   ← load_data.bit[  0:127]
VSR[32×TX+2×Tp+1] ← load_data.bit[128:255]
```

Let XTp be the value 32×TX + 2×Tp (i.e., only even values of XTp can be encoded in the instruction).

Let the effective address (EA) be the sum of the integer value in GPR[RA], or 0 if RA=0, and the integer value in GPR[RB].

When Big-Endian byte ordering is employed, the contents of the octword in storage at address EA are placed into load_data in such an order that;

- the contents of the byte in storage at address EA are placed into byte 0 of load_data,
- the contents of the byte in storage at address EA+1 are placed into byte 1 of load_data, and so forth until
- the contents of the byte in storage at address EA+31 are placed into byte 31 of load_data.

When Little-Endian byte ordering is employed, the contents of the octword in storage at address EA are placed into load_data in such an order that;

- the contents of the byte in storage at address EA are placed into byte 31 of load_data,
- the contents of the byte in storage at address EA+1 are placed into byte 30 of load_data, and so forth until
- the contents of the byte in storage at address EA+31 are placed into byte 0 of load_data.

Bits 0-127 of load_data are placed into VSR[XTp].

Bits 128-255 of load_data is placed into VSR[XTp+1].

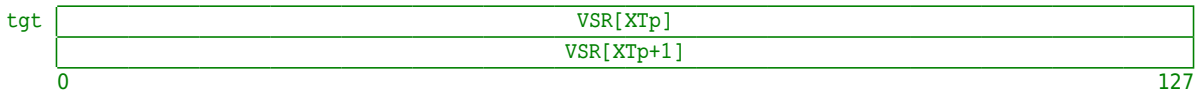
Special Registers Altered:

None

Programming Note

For best performance, EA should be word-aligned.

VSR Data Layout for lxvpx



Store VSX Vector Paired DQ-form

stxvp XSp,disp(RA)

6	Sp	SX	RA	DQ	1
0	6	10	11	16	28 31

Prefixed Store VSX Vector Paired 8LS:D-form

pstxvp XSp,D(RA),R

Prefix:

1	0	0	//	R	//	d0
0	6	8	9	11	12	14 31

Suffix:

62	Sp	SX	RA	d1
0	6	10	11	16 31

```
if MSR.VSX=0 then VSX_Unavailable()

EAbase ← (RA=0) ? 0 : GPR[RA]
if ''stxvp'' then
   EAdisp ← EXTS64(DQ || 0b0000)
if ''pstxvp'' then
   EAdisp ← EXTS64(d0 || d1)

if ''stxvp''        then EA ← EAbase + EAdisp
if ''pstxvp'' & R=0 then EA ← EAbase + EAdisp
if ''pstxvp'' & R=1 then EA ← CIA    + EAdisp

store_data.bit[  0:127] ← VSR[32×SX+2×Sp]
store_data.bit[128:255] ← VSR[32×SX+2×Sp+1]

MEM(EA,32) ← store_data
```

Let XSp be the value 32×SX + 2×Sp (i.e., only even values of XSp can be encoded in the instruction).

For ***stxvp***, let the effective address (EA) be the sum of the integer value in GPR[RA], or 0 if RA=0 and the value DQ||0b0000, sign-extended to 64 bits.

For *pstxvp*, if R=0, let the effective address (EA) be the sum of the integer value in GPR[RA], or 0 if RA=0, and the value d0||d1, sign-extended to 64 bits.

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

Let store_data be the contents of VSR[XSp] concatenated with VSR[XSp+1].

When Big-Endian byte ordering is employed, store_data is placed into the octword in storage at address EA in such an order that;

- byte 0 of store_data is placed into the byte in storage at address EA,
- byte 1 of store_data is placed into the byte in storage at address EA+1, and so forth until
- byte 31 of store_data is placed into the byte in storage at address EA+31.

When Little-Endian byte ordering is employed, store_data is placed into the octword in storage at address EA in such an order that;

- byte 0 of store_data is placed into the byte in storage at address EA+31,
- byte 1 of store_data is placed into the byte in storage at address EA+30, and so forth until
- byte 31 of store_data is placed into the byte in storage at address EA.

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

Special Registers Altered:

None

Extended Mnemonics:

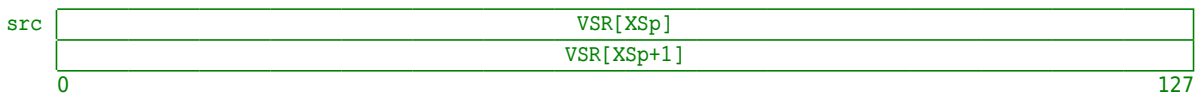
Extended Mnemonics for *Store VSX Vector Paired Prefixed*:

Extended mnemonic:		Equivalent to:	
pstxvp	XSp,D(RA)	pstxvp	XSp,D(RA),0
pstxvp	XSp,D	pstxvp	XSp,D(0),1

Programming Note

For best performance, EA should be word-aligned.

VSR Data Layout for [p]stxvp



Store VSX Vector Paired Indexed X-form

stxvpx XSp,RA,RB

31	Sp	SX	RA	RB	461	/
0	6	10	11	16	21	31

```
if MSR.VSX=0 then VSX_Unavailable()

EA ← ((RA=0) ? 0 : GPR[RA]) + GPR[RB]

store_data.bit[  0:127] ← VSR[32×SX+2×Sp]
store_data.bit[128:255] ← VSR[32×SX+2×Sp+1]

MEM(EA,32) ← store_data
```

Let XSp be the value 32×SX + 2×Sp (i.e., only even values of XSp can be encoded in the instruction).

Let the effective address (EA) be the sum of the integer value in GPR[RA], or 0 if RA=0, and the integer value in GPR[RB].

Let store_data be the contents of VSR[XSp] concatenated with VSR[XSp+1].

When Big-Endian byte ordering is employed, store_data is placed into the octword in storage at address EA in such an order that;

- byte 0 of store_data is placed into the byte in storage at address EA,
- byte 1 of store_data is placed into the byte in storage at address EA+1, and so forth until
- byte 31 of store_data is placed into the byte in storage at address EA+31.

When Little-Endian byte ordering is employed, store_data is placed into the octword in storage at address EA in such an order that;

- byte 0 of store_data is placed into the byte in storage at address EA+31,
- byte 1 of store_data is placed into the byte in storage at address EA+30, and so forth until
- byte 31 of store_data is placed into the byte in storage at address EA.

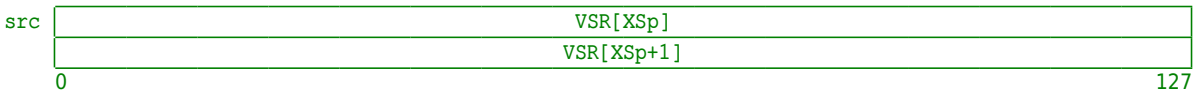
Special Registers Altered:

None

Programming Note

For best performance, EA should be word-aligned.

VSR Data Layout for stxvpx



7.6.2.2 VSX Binary Floating-Point Sign Manipulation Instructions

7.6.2.2.1 VSX Scalar Binary Floating-Point Sign Manipulation Instructions

VSX Scalar Absolute Double-Precision XX2-form

xsabsdp XT,XB

60	T	///	B	345	BX	TX
0	6	11	16	21	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

src ← VSR[32×BX+B].dword[0]
VSR[32×TX+T].dword[0] ← bfp64_ABSOLUTE(src)
VSR[32×TX+T].dword[1] ← 0x0000_0000_0000_0000
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

The absolute value of the double-precision floating-point operand in doubleword element 0 of VSR[XB] is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are set to 0.

Special Registers Altered:

None

Programming Note

This instruction can be used to operate on a single-precision source operand.

Programming Note

Previous versions of the architecture allowed the contents of doubleword 1 of the result register to be undefined. However, all processors that support this instruction write 0s into doubleword 1 of the result register, as is required by this version of the architecture.

VSX Scalar Absolute Quad-Precision X-form

xsabsqp VRT,VRB

63	VRT	0	VRB	804	/
0	6	11	16	21	31

```
if MSR.VSX=0 then VSX_Unavailable()

VSR[VRT+32] ← bfp128_ABSOLUTE(VSR[VRB+32])
```

Let XT be the value VRT + 32.

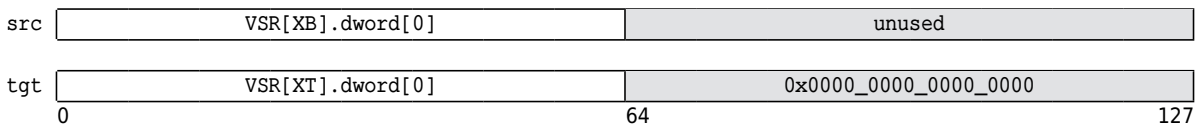
Let XB be the value VRB + 32.

The absolute value of the quad-precision floating-point value in VSR[XB] is placed into VSR[XT].

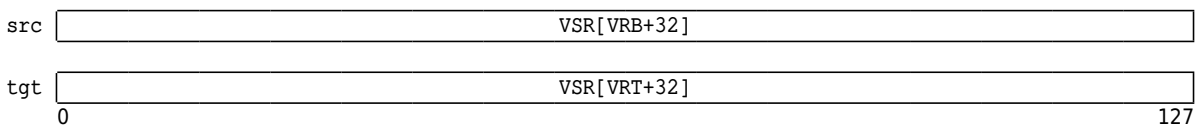
Special Registers Altered:

None

VSR Data Layout for xsabsdp



VSR Data Layout for xsabsqp



VSX Scalar Copy Sign Double-Precision XX3-form

xscpsgndp XT,XA,XB

60	T	A	B	176	AX	BX	TX
0	6	11	16	21	29	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

src1 ← VSR[32×AX+A].dword[0] & 0x8000_0000_0000_0000
src2 ← VSR[32×BX+B].dword[0] & 0x7FFF_FFFF_FFFF_FFFF
VSR[32×TX+T].dword[0] ← src1 | src2
VSR[32×TX+T].dword[1] ← 0x0000_0000_0000_0000
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

Bit 0 of VSR[XT] is set to the contents of bit 0 of VSR[XA].

Bits 1:63 of VSR[XT] are set to the contents of bits 1:63 of VSR[XB].

The contents of doubleword element 1 of VSR[XT] are set to 0.

Special Registers Altered:

None

Programming Note

This instruction can be used to operate on single-precision source operands.

Programming Note

Previous versions of the architecture allowed the contents of doubleword 1 of the result register to be undefined. However, all processors that support this instruction write 0s into doubleword 1 of the result register, as is required by this version of the architecture.

VSX Scalar Copy Sign Quad-Precision X-form

xscpsgnqp VRT,VRA,VRB

63	VRT	VRA	VRB	100	/
0	6	11	16	21	31

```
if MSR.VSX=0 then VSX_Unavailable()

src1 ← VSR[VRA+32] &
       0x8000_0000_0000_0000_0000_0000_0000_0000
src2 ← VSR[VRB+32] &
       0x7FFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF

VSR[VRT+32] ← src1 | src2
```

Let src1 be the floating-point value in VSR[VRA+32] represented in quad-precision format.

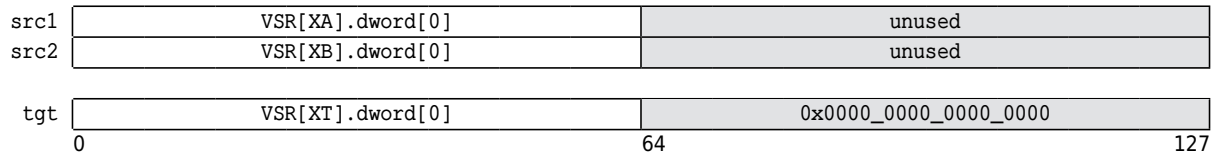
Let src2 be the floating-point value in VSR[VRB+32] represented in quad-precision format.

src2 is placed into VSR[VRT+32] with the sign of src1.

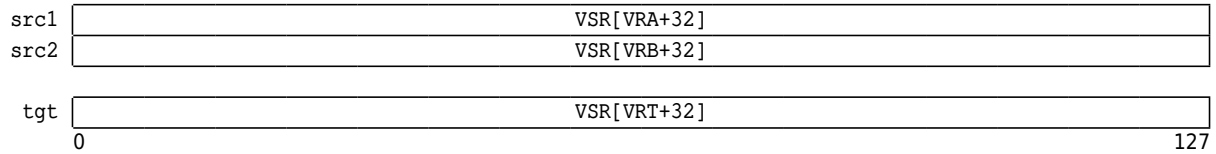
Special Registers Altered:

None

VSR Data Layout for xscpsgndp



VSR Data Layout for xscpsgnqp



VSX Scalar Negative Absolute Double-Precision XX2-form

xsnabsdp XT,XB

60	T	///	B	361	BX	TX
0	6	11	16	21	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

src ← VSR[32×BX+B].dword[0]
VSR[32×TX+T].dword[0] ← bfp64_NEGATIVE_ABSOLUTE(src)
VSR[32×TX+T].dword[1] ← 0x0000_0000_0000_0000
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

The contents of doubleword element 0 of VSR[XB], with bit 0 set to 1, is placed into doubleword element 0 of VSR[XT].

The contents of doubleword element 1 of VSR[XT] are set to 0.

Special Registers Altered:

None

Programming Note

This instruction can be used to operate on a single-precision source operand.

Programming Note

Previous versions of the architecture allowed the contents of doubleword 1 of the result register to be undefined. However, all processors that support this instruction write 0s into doubleword 1 of the result register, as is required by this version of the architecture.

VSX Scalar Negative Absolute Quad-Precision X-form

xsnabsqp VRT,VRB

63	VRT	8	VRB	804	TX
0	6	11	16	21	31

```
if MSR.VSX=0 then VSX_Unavailable()

VSR[VRT+32] ← bfp128_NEGATIVE_ABSOLUTE(VSR[VRB+32])
```

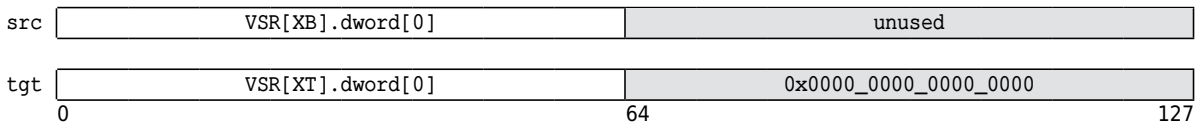
Let src be the floating-point value in VSR[VRB+32] represented in quad-precision format.

The negative absolute value of src is placed into VSR[VRT+32] in quad-precision format.

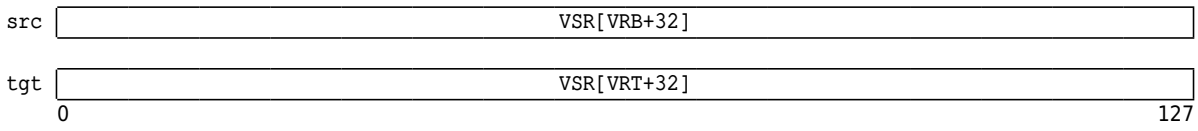
Special Registers Altered:

None

VSR Data Layout for xsnabsdp



VSR Data Layout for xsnabsqp



VSX Scalar Negate Double-Precision XX2-form

xsnegdp XT,XB

60	T	///	B	377	BX	TX
0	6	11	16	21	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

src ← VSR[32×BX+B].dword[0]
VSR[32×TX+T].dword[0] ← bfp64_NEGATE(src)
VSR[32×TX+T].dword[1] ← 0x0000_0000_0000_0000
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

The contents of doubleword element 0 of VSR[XB], with bit 0 complemented, is placed into doubleword element 0 of VSR[XT].

The contents of doubleword element 1 of VSR[XT] are set to 0.

Special Registers Altered:

None

Programming Note

This instruction can be used to operate on a single-precision source operand.

Programming Note

Previous versions of the architecture allowed the contents of doubleword 1 of the result register to be undefined. However, all processors that support this instruction write 0s into doubleword 1 of the result register, as is required by this version of the architecture.

VSX Scalar Negate Quad-Precision X-form

xsnegqp VRT,VRB

63	VRT	16	VRB	804	/
0	6	11	16	21	31

```
if MSR.VSX=0 then VSX_Unavailable()

VSR[VRT+32] ← bfp128_NEGATE(VSR[VRB+32])
```

Let src be the floating-point value in VSR[VRB+32] represented in quad-precision format.

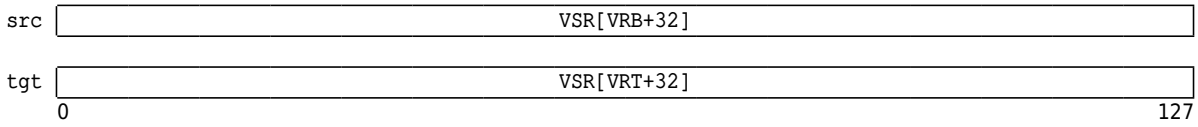
src is negated and placed into VSR[VRT+32] in quad-precision format.

Special Registers Altered:

None

VSR Data Layout for xsnegdp

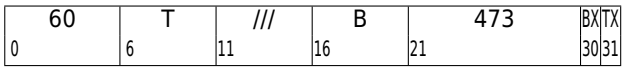
VSR Data Layout for xsnegqp



7.6.2.2.2 VSX Vector Binary Floating-Point Sign Manipulation Instructions

VSX Vector Absolute Double-Precision XX2-form

xvabsdp XT,XB




```
if MSR.VSX=0 then VSX_Unavailable()

do i = 0 to 1
   src ← VSR[32×BX+B].dword[i]
   VSR[32×TX+T].dword[i] ← bfp64_ABSOLUTE(src)
end
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

For each integer value i from 0 to 1, do the following.

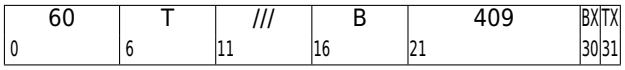
The contents of doubleword element i of VSR[XB], with bit 0 set to 0, is placed into doubleword element i of VSR[XT].

Special Registers Altered:

None

VSX Vector Absolute Single-Precision XX2-form

xvabssp XT,XB




```
if MSR.VSX=0 then VSX_Unavailable()

do i = 0 to 3
   src ← VSR[32×BX+B].word[i]
   VSR[32×TX+T].word[i] ← bfp32_ABSOLUTE(src)
end
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

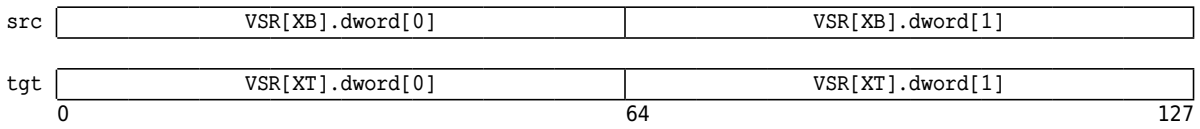
For each integer value i from 0 to 3, do the following.

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

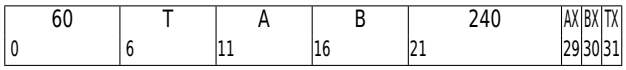
VSR Data Layout for xvabsdp



VSR Data Layout for xvabssp

VSX Vector Copy Sign Double-Precision XX3-form

xvcpsgndp XT,XA,XB




```
if MSR.VSX=0 then VSX_Unavailable()

do i = 0 to 1
   src1 ← VSR[32×AX+A].dword[i] & 0x8000_0000_0000_0000
   src2 ← VSR[32×BX+B].dword[i] & 0x7FFF_FFFF_FFFF_FFFF
   VSR[32×TX+T].dword[i] ← src1 | src2
end
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

For each integer value i from 0 to 1, do the following.

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

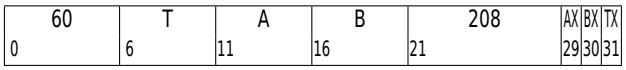
Extended Mnemonics:

Extended Mnemonics for *VSX Vector Copy Sign Double-Precision*:

Extended mnemonic:	Equivalent to:
xvmovdp XT,XB	xvcpsgndpXT,XB,XB

VSX Vector Copy Sign Single-Precision XX3-form

xvcpsgnsp XT,XA,XB




```
if MSR.VSX=0 then VSX_Unavailable()

do i = 0 to 3
   src1 ← VSR[32×AX+A].word[i] & 0x8000_0000
   src2 ← VSR[32×BX+B].word[i] & 0x7FFF_FFFF
   VSR[32×TX+T].word[i] ← src1 | src2
end
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

For each integer value i from 0 to 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:

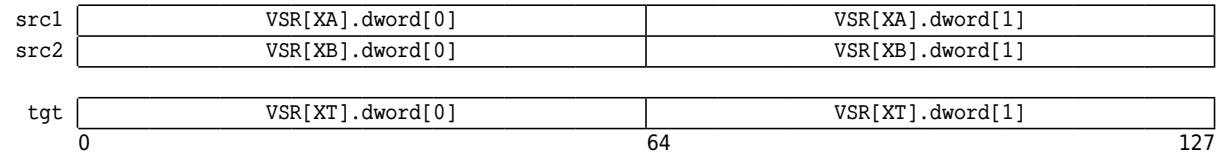
None

Extended Mnemonics:

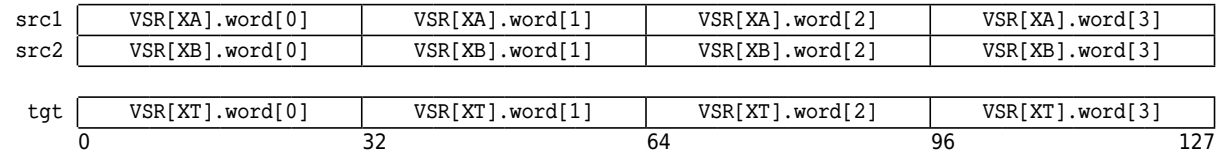
Extended Mnemonics for *VSX Vector Copy Sign Single-Precision*:

Extended mnemonic:	Equivalent to:
xvmovsp XT,XB	xvcpsgnspXT,XB,XB

VSR Data Layout for xvcpsgndp



VSR Data Layout for xvcpsgnsp



VSX Vector Negative Absolute Double-Precision XX2-form

xvnabsdp XT,XB

60	T	///	B	489	BX	TX
0	6	11	16	21	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

do i = 0 to 1
   src ← VSR[32×BX+B].dword[i]
   VSR[32×TX+T].dword[i] ← bfp64_NEGATIVE_ABSOLUTE(src)
end
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

For each integer value i from 0 to 1, do the following.

The contents of doubleword element i of VSR[XB], with bit 0 set to 1, is placed into doubleword element i of VSR[XT].

Special Registers Altered:

None

VSX Vector Negative Absolute Single-Precision XX2-form

xvnabssp XT,XB

60	T	///	B	425	BX	TX
0	6	11	16	21	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

do i = 0 to 3
   src ← VSR[32×BX+B].word[i]
   VSR[32×TX+T].word[i] ← bfp32_NEGATIVE_ABSOLUTE(src)
end
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

For each integer value i from 0 to 3, do the following.

The contents of word element i of VSR[XB], with bit 0 set to 1, is placed into word element i of VSR[XT].

Special Registers Altered:

None

VSR Data Layout for xvnabsdp

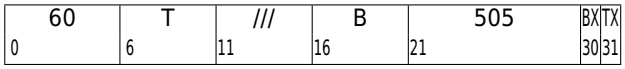
src	VSR[XB].dword[0]	VSR[XB].dword[1]
tgt	VSR[XT].dword[0]	VSR[XT].dword[1]
	0	64 127

VSR Data Layout for xvnabssp

src	VSR[XB].word[0]	VSR[XB].word[1]	VSR[XB].word[2]	VSR[XB].word[3]
tgt	VSR[XT].word[0]	VSR[XT].word[1]	VSR[XT].word[2]	VSR[XT].word[3]
	0	32	64	96 127

VSX Vector Negate Double-Precision XX2-form

xvnegdp XT,XB




```
if MSR.VSX=0 then VSX_Unavailable()

do i = 0 to 1
   src ← VSR[32×BX+B].dword[i]
   VSR[32×TX+T].dword[i] ← bfp64_NEGATE(src)
end
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

For each integer value i from 0 to 1, do the following.

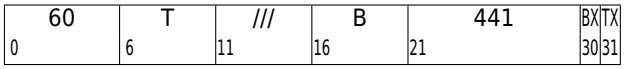
The contents of doubleword element i of VSR[XB], with bit 0 complemented, is placed into doubleword element i of VSR[XT].

Special Registers Altered:

None

VSX Vector Negate Single-Precision XX2-form

xvnegsp XT,XB




```
if MSR.VSX=0 then VSX_Unavailable()

do i = 0 to 3
   src ← VSR[32×BX+B].word[i]
   VSR[32×TX+T].word[i] ← bfp32_NEGATE(src)
end
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

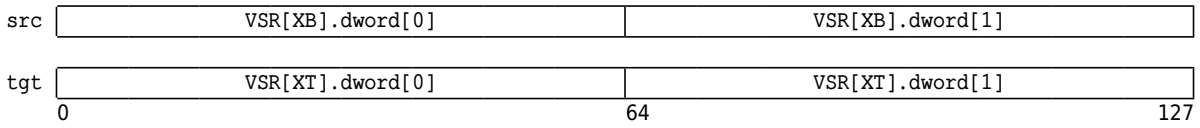
For each integer value i from 0 to 3, do the following.

The contents of word element i of VSR[XB], with bit 0 complemented, is placed into word element i of VSR[XT].

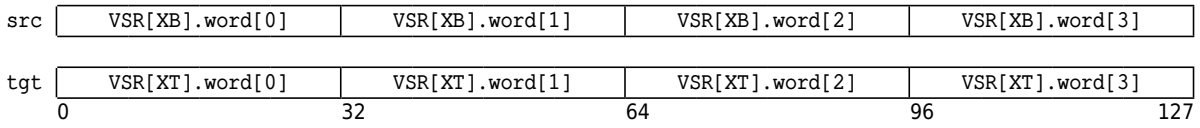
Special Registers Altered:

None

VSR Data Layout for xvnegdp



VSR Data Layout for xvnegsp



7.6.2.3 VSX Binary Floating-Point Arithmetic Instructions

7.6.2.3.1 VSX Scalar Binary Floating-Point Arithmetic Instructions

VSX Scalar Add Double-Precision XX3-form

xsadddp XT,XA,XB

60	T	A	B	32	AX	BX	TX
0	6	11	16	21	29	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

reset_xflags()

src1   ← bfp_CONVERT_FROM_BFP64(VSR[VRA+32].dword[0])
src2   ← bfp_CONVERT_FROM_BFP64(VSR[VRB+32].dword[0])
v      ← bfp_ADD(src1, src2)
rnd    ← bfp_ROUND_TO_BFP64(0b0,FPSCR.RN, v)
result ← bfp64_CONVERT_FROM_BFP(rnd)

if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
if vxisi_flag=1 then SetFX(FPSCR.VXISI)
if ox_flag=1 then SetFX(FPSCR.OX)
if ux_flag=1 then SetFX(FPSCR.UX)
if xx_flag=1 then SetFX(FPSCR.XX)

vx_flag  ← vxsnan_flag | vxisi_flag
vex_flag ← FPSCR.VE & vx_flag

if vex_flag=0 then do
   VSR[32×TX+T].dword[0] ← result
   VSR[32×TX+T].dword[1] ← 0x0000_0000_0000_0000
   FPSCR.FPRF ← fprf_CLASS_BFP64(result)
   FPSCR.FR   ← inc_flag
   FPSCR.FI   ← xx_flag
end
else do
   FPSCR.FR   ← 0b0
   FPSCR.FI   ← 0b0
end
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

Let src1 be the double-precision floating-point value in doubleword element 0 of VSR[XA].

Let src2 be the double-precision floating-point value in doubleword element 0 of VSR[XB].

src2 is added[15] to src1, producing a sum having unbounded range and precision.

The sum is normalized[16].

See Table 7.8, “Actions for xsadddp,” on page 616.

The intermediate result is rounded to double-precision using the rounding mode specified by RN.

See Table 7.9, “Scalar Floating-Point Intermediate Result Handling,” on page 617.

The result is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are set to 0.

FPRF is set to the class and sign of the result. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.

If a trap-enabled invalid operation exception occurs, VSR[XT] and FPRF are not modified, and FR and FI are set to 0.

See Table 7.10, “VSX Scalar Floating-Point Final Result,” on page 618.

Special Registers Altered:

Register	*Field(s)*
FPSCR	FPRF FR FI FX OX UX XX VXSNAN VXISI

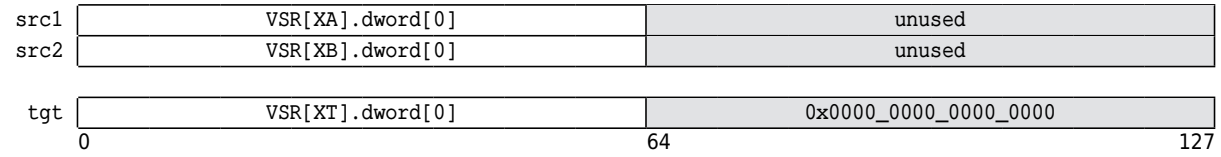
Programming Note

Previous versions of the architecture allowed the contents of doubleword 1 of the result register to be undefined. However, all processors that support this instruction write 0s into doubleword 1 of the result register, as is required by this version of the architecture.

[15] 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.

[16] 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 xsadddp



		src2							
		−Infinity	**−NZF**	**−Zero**	**+Zero**	**+NZF**	**+Infinity**	**QNaN**	**SNaN**
src1	**−Infinity**	v ← −Infinity	v ← −Infinity	v ← −Infinity	v ← −Infinity	v ← −Infinity	v ← dQNaN vxisi_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	−NZF	v ← −Infinity	v ← A(src1,src2)	v ← src1	v ← src1	v ← A(src1,src2)	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	−Zero	v ← −Infinity	v ← src2	v ← −Zero	v ← Rezd	v ← src2	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+Zero	v ← −Infinity	v ← src2	v ← Rezd	v ← +Zero	v ← src2	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+NZF	v ← −Infinity	v ← A(src1,src2)	v ← src1	v ← src1	v ← A(src1,src2)	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+Infinity	v ← dQNaN vxisi_flag ← 1	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	QNaN	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1 vxsnan_flag ← 1
	SNaN	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

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 signficand precision and unbounded exponent range.

Table 7.8: Actions for xsadddp

Table 7.9: Scalar Floating-Point Intermediate Result Handling

Range of v	Case	Rounding Mode				
		Round To Nearest (RTN)	Round Towards Zero (RTZ)	Round Towards +Infinity (RTP)	Round Towards −Infinity (RTM)	Round To Odd (RTO)
v is a QNaN	Special	r ← v	r ← v	r ← v	r ← v	r ← v
v = −Infinity	Special	r ← v	r ← v	r ← v	r ← v	r ← v
−Infinity < v ≤ −(Nmax + 1ulp)	Overflow	q ← rnd(v) r ← −Infinity	q ← rnd(v) r ← -Nmax	q ← rnd(v) r ← -Nmax	q ← rnd(v) r ← −Infinity	q ← rnd(v) r ← -Nmax
−(Nmax + 1ulp) < v ≤ −(Nmax + $\frac{1}{2}$ulp)	Overflow	q ← rnd(v) r ← −Infinity	—	—	q ← rnd(v) r ← −Infinity	—
	Normal	—	r ← −Nmax	r ← −Nmax	—	r ← −Nmax
−(Nmax + $\frac{1}{2}$ulp) < v < -Nmax	Overflow	—	—	—	q ← rnd(v) r ← −Infinity	—
	Normal	r ← −Nmax	r ← −Nmax	r ← −Nmax	—	r ← −Nmax
−Nmax ≤ v ≤ −Nmin	Normal	r ← rnd(v)	r ← rnd(v)	r ← rnd(v)	r ← rnd(v)	r ← rnd(v)
−Nmin < v < -Zero	Tiny	q ← rnd(v) r ← rnd(den(v))	q ← rnd(v) r ← rnd(den(v))	q ← rnd(v) r ← rnd(den(v))	q ← rnd(v) r ← rnd(den(v))	q ← rnd(v) r ← rnd(den(v))
v = −Zero	Special	r ← v	r ← v	r ← v	r ← v	r ← v
v = Rezd	Special	r ← +Zero	r ← +Zero	r ← +Zero	r ← −Zero	r ← +Zero (RN=RTN) r ← +Zero (RN=RTZ) r ← +Zero (RN=RTP) r ← -Zero (RN=RTM)
v = +Zero	Special	r ← v	r ← v	r ← v	r ← v	r ← v
+Zero < v < +Nmin	Tiny	q ← rnd(v) r ← rnd(den(v))	q ← rnd(v) r ← rnd(den(v))	q ← rnd(v) r ← rnd(den(v))	q ← rnd(v) r ← rnd(den(v))	q ← rnd(v) r ← rnd(den(v))
+Nmin ≤ v ≤ +Nmax	Normal	r ← rnd(v)	r ← rnd(v)	r ← rnd(v)	r ← rnd(v)	r ← rnd(v)
+Nmax < v < +(Nmax + $\frac{1}{2}$ulp)	Overflow	—	—	q ← rnd(v) r ← +Infinity	—	—
	Normal	r ← +Nmax	r ← +Nmax	—	r ← +Nmax	r ← +Nmax
+(Nmax + $\frac{1}{2}$ulp) ≤ v < +(Nmax + 1ulp)	Overflow	q ← rnd(v) r ← +Infinity	—	q ← rnd(v) r ← +Infinity	—	—
	Normal	—	r ← +Nmax	—	r ← +Nmax	r ← +Nmax
+(Nmax + 1ulp) ≤ v < +Infinity	Overflow	q ← rnd(v) r ← +Infinity	q ← rnd(v) r ← +Nmax	q ← rnd(v) r ← +Infinity	q ← rnd(v) r ← +Nmax	q ← rnd(v) r ← +Nmax
v = +Infinity	Special	r ← v	r ← v	r ← v	r ← v	r ← v

Explanation:

— This situation cannot occur.

v The precise intermediate result defined in the instruction having unbounded range and precision.

den(x) The significand of x is shifted right by the amount of the difference between the target rounding precision Emin and the unbiased exponent of x. The unbiased exponent of the denormalized value is Emin. The significand of the denormalized value has unbounded significand precision.
Emin = −16382 (quad-precision)
Emin = −16382 (double-extended-precision)
Emin = −1022 (double-precision)
Emin = −126 (single-precision)

Rezd Exact-zero-difference result. Applies only to add operations involving source operands having the same magnitude and different signs or subtract operations involving source operands having the same magnitude and same signs. Whether +Zero or -Zero is returned is controlled by the setting of the rounding mode in RN, even when the rounding mode is overridden to Round to Odd.

rnd(x) The significand of x is rounded to the target rounding precision according to the rounding mode specified in FPSCR.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.
Nmax = ± 2^{+16383} × 1.FFFFFFFFFFFFFFFFFFFFFFFFFFFF (quad-precision)
Nmax = ± 2^{+16383} × 1.FFFFFFFFFFFFFFFF000000000000 (double-extended-precision)
Nmax = ± 2^{+1023} × 1.FFFFFFFFFFFFF0000000000000000 (double-precision)
Nmax = ± 2^{+127} × 1.FFFFFF00000000000000000000 (single-precision)

Nmin Smallest (in magnitude) representable normalized number in the target rounding precision format.
Nmin = ± 2^{-16382} × 1.00000000000000000000000000000 (quad-precision)
Nmin = ± 2^{-16382} × 1.00000000000000000000000000000 (double-extended-precision)
Nmin = ± 2^{-1022} × 1.00000000000000000000000000000 (double-precision)
Nmin = ± 2^{-126} × 1.00000000000000000000000000000 (single-precision)

ulp Least significant bit in the target precision format's significand (Unit in the Last Position).

Table 7.10: VSX Scalar Floating-Point Final Result

Case	FPSCR.VE	FPSCR.OE	FPSCR.UE	FPSCR.ZE	FPSCR.XE	vxsnan_flag	vximz_flag	vxisi_flag	vxidi_flag	vxzdz_flag	vxsqrt_flag	zx_flag	Is r inexact? ($r \neq v$)	Is r incremented? ($\lvert r\rvert > \lvert v\rvert$)	Is q inexact? ($q \neq v$)	Is q incremented? ($\lvert q\rvert > \lvert v\rvert$)	Returned Results and Status Setting
Special	–	–	–	0	–	–	–	–	–	–	–	1	–	–	–	–	T(r), class_bfp(r), fi(0), fr(0), fx(ZX)
	–	–	–	1	–	–	–	–	–	–	–	1	–	–	–	–	fx(ZX), error()
	0	–	–	–	–	–	–	–	–	–	1	–	–	–	–	–	T(r), class_bfp(r), fi(0), fr(0), fx(VXSQRT)
	0	–	–	–	–	–	–	–	–	1	–	–	–	–	–	–	T(r), class_bfp(r), fi(0), fr(0), fx(VXZDZ)
	0	–	–	–	–	–	–	–	1	–	–	–	–	–	–	–	T(r), class_bfp(r), fi(0), fr(0), fx(VXIDI)
	0	–	–	–	–	–	–	1	–	–	–	–	–	–	–	–	T(r), class_bfp(r), fi(0), fr(0), fx(VXISI)
	0	–	–	–	–	0	1	–	–	–	–	–	–	–	–	–	T(r), class_bfp(r), fi(0), fr(0), fx(VXIMZ)
	0	–	–	–	–	1	0	–	–	–	–	–	–	–	–	–	T(r), class_bfp(r), fi(0), fr(0), fx(VXSNAN)
	0	–	–	–	–	1	0	–	–	–	–	–	–	–	–	–	T(r), class_bfp(r), fi(0), fr(0), fx(VXSNAN), fx(VXIMZ)
	1	–	–	–	–	–	–	–	–	–	1	–	–	–	–	–	fx(VXSQRT), error()
	1	–	–	–	–	–	–	–	–	1	–	–	–	–	–	–	fx(VXZDZ), error()
	1	–	–	–	–	–	–	–	1	–	–	–	–	–	–	–	fx(VXIDI), error()
	1	–	–	–	–	–	–	1	–	–	–	–	–	–	–	–	fx(VXISI), error()
	1	–	–	–	–	0	1	–	–	–	–	–	–	–	–	–	fx(VXIMZ), error()
	1	–	–	–	–	1	0	–	–	–	–	–	–	–	–	–	fx(VXSNAN), error()
	1	–	–	–	–	1	1	–	–	–	–	–	–	–	–	–	fx(VXSNAN), fx(VXIMZ), error()
Normal	–	–	–	–	–	0	0	0	0	0	0	0	no	–	–	–	T(r), class_bfp(r), fi(0), fr(0)
	–	–	–	–	0	0	0	0	0	0	0	0	yes	no	–	–	T(r), class_bfp(r), fi(1), fr(0), fx(XX)
	–	–	–	–	0	0	0	0	0	0	0	0	yes	yes	–	–	T(r), class_bfp(r), fi(1), fr(1), fx(XX)
	–	–	–	–	1	0	0	0	0	0	0	0	yes	no	–	–	T(r), class_bfp(r), fi(1), fr(0), fx(XX), error()
	–	–	–	–	1	0	0	0	0	0	0	0	yes	yes	–	–	T(r), class_bfp(r), fi(1), fr(1), fx(XX), error()
Overflow	–	0	–	–	0	0	0	0	0	0	0	0	–	–	–	–	T(r), class_bfp(r), fi(1), fr(?), fx(OX), fx(XX)
	–	0	–	–	1	0	0	0	0	0	0	0	–	–	–	–	T(r), class_bfp(r), fi(1), fr(?), fx(OX), fx(XX), error()
	–	1	–	–	–	0	0	0	0	0	0	0	–	–	no	–	T(q÷β), class_bfp(q÷β), fi(0), fr(0), fx(OX), error()
	–	1	–	–	–	0	0	0	0	0	0	0	–	–	yes	no	T(q÷β), class_bfp(q÷β), fi(1), fr(0), fx(OX), fx(XX), error()
	–	1	–	–	–	0	0	0	0	0	0	0	–	–	yes	yes	T(q÷β), class_bfp(q÷β), fi(1), fr(1), fx(OX), fx(XX), error()
Tiny	–	–	0	–	–	0	0	0	0	0	0	0	no	–	–	–	T(r), class_bfp(r), fi(0), fr(0)
	–	–	0	–	0	0	0	0	0	0	0	0	yes	no	–	–	T(r), class_bfp(r), fi(1), fr(0), fx(UX), fx(XX)
	–	–	0	–	0	0	0	0	0	0	0	0	yes	yes	–	–	T(r), class_bfp(r), fi(1), fr(1), fx(UX), fx(XX)
	–	–	0	–	1	0	0	0	0	0	0	0	yes	no	–	–	T(r), class_bfp(r), fi(1), fr(0), fx(UX), fx(XX), error()
	–	–	0	–	1	0	0	0	0	0	0	0	yes	yes	–	–	T(r), class_bfp(r), fi(1), fr(1), fx(UX), fx(XX), error()
	–	–	1	–	–	0	0	0	0	0	0	0	–	–	no	–	T(q×β), class_bfp(q×β), fi(0), fr(0), fx(UX), error()
	–	–	1	–	–	0	0	0	0	0	0	0	–	–	yes	no	T(q×β), class_bfp(q×β), fi(1), fr(0), fx(UX), fx(XX), error()
	–	–	1	–	–	0	0	0	0	0	0	0	–	–	yes	yes	T(q×β), class_bfp(q×β), fi(1), fr(1), fx(UX), fx(XX), error()

Table 7.10: VSX Scalar Floating-Point Final Result (continued)

Explanation:

–	The results do not depend on this condition.
T(x)	Places the result into the target VSR. For scalar single-precision and double-precision results VSR[XT].dword[0] ← bfp64_CONVERT_FROM_BFP(r) VSR[XT].dword[1] ← 0x0000_0000_0000_0000 For scalar quad-precision results VSR[VRT+32] ← bfp128_CONVERT_FROM_BFP(r)
class_bfp(x)	Sets FPSCR.FPRF to the sign and class of x. FPSCR.FPRF ← fprf_CLASS_BFP32(x) (single-precision) FPSCR.FPRF ← fprf_CLASS_BFP64(x) (double-precision) FPSCR.FPRF ← fprf_CLASS_BFP128(x) (quad-precision)
fx(x)	FPSCR.FX is set to 1 if FPSCR.x=0. 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.
β	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 518 for trap-enabled Overflow exceptions. See Table 7.4.4.2, “Action for UE=1,” on page 521 for trap-enabled Underflow exceptions.
q	The value defined in Table 7.9, “Scalar Floating-Point Intermediate Result Handling,” on page 617, significand rounded to the target rounding precision, unbounded exponent range.
r	The value defined in Table 7.9, “Scalar Floating-Point Intermediate Result Handling,” on page 617, 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 Quad-Precision [using round to Odd] X-form

xsaddqp	VRT,VRA,VRB	(RO=0)
xsaddqpo	VRT,VRA,VRB	(RO=1)

63	VRT	VRA	VRB	4	RO
0	6	11	16	21	31

```
if MSR.VSX=0 then VSX_Unavailable()

reset_xflags()

src1   ← bfp_CONVERT_FROM_BFP128(VSR[VRA+32])
src2   ← bfp_CONVERT_FROM_BFP128(VSR[VRB+32])
v      ← bfp_ADD(src1, src2)
rnd    ← bfp_ROUND_TO_BFP128(RO, FPSCR.RN, v)
result ← bfp128_CONVERT_FROM_BFP(rnd)

if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
if vxisi_flag=1  then SetFX(FPSCR.VXISI)
if ox_flag=1     then SetFX(FPSCR.OX)
if ux_flag=1     then SetFX(FPSCR.UX)
if xx_flag=1     then SetFX(FPSCR.XX)

vx_flag  ← vxsnan_flag | vxisi_flag
vex_flag ← FPSCR.VE & vx_flag

if vex_flag=0 then do
   VSR[VRT+32] ← result
   FPSCR.FPRF  ← fprf_CLASS_BFP128(result)
end
FPSCR.FR ← (vx_flag=0) & inc_flag
FPSCR.FI ← (vx_flag=0) & xx_flag
```

Let src1 be the floating-point value in VSR[VRA+32] represented in quad-precision format.

Let src2 be the floating-point value in VSR[VRB+32] represented in quad-precision format.

If either src1 or src2 is a Signalling NaN, an Invalid Operation exception occurs and VXSNAN is set to 1.

If src1 and src2 are Infinity values having opposite signs, an Invalid Operation exception occurs and VXISI is set to 1.

If src1 is a Signalling NaN, the result is the Quiet NaN corresponding to src1.

Otherwise, if src1 is a Quiet NaN, the result is src1.

Otherwise, if src2 is a Signalling NaN, the result is the Quiet NaN corresponding to src2.

Otherwise, if src2 is a Quiet NaN, the result is src2.

Otherwise, if src1 and src2 are Infinity values having opposite signs, the result is the default Quiet NaN[17].

Otherwise, do the following.

The normalized sum of src2 added to src1 is produced with unbounded significand precision and exponent range.

See Table 7.11, "Actions for xsaddqp[o]," on page 621.

If the intermediate result is *Tiny* (i.e., the unbiased exponent is less than −16382) and UE=0, the significand is shifted right N bits, where N is the difference between −16382 and the unbiased exponent of the intermediate result. The exponent of the intermediate result is set to the value −16382.

If RO=1, let the rounding mode be Round to Odd. Otherwise, let the rounding mode be specified by RN. The intermediate result is rounded to quad-precision using the specified rounding mode.

See Table 7.9, "Scalar Floating-Point Intermediate Result Handling," on page 617.

The result is placed into VSR[VRT+32] in quad-precision format.

FPRF is set to the class and sign of the result. FR is set to indicate if the rounded result was incremented. FI is set to indicate the result is inexact.

If a trap-disabled Invalid Operation exception occurs, FR and FI are set to 0.

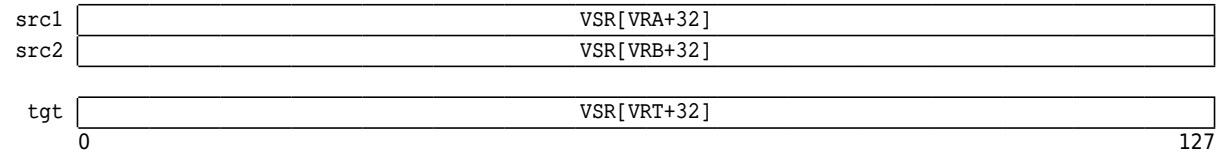
If a trap-enabled Invalid Operation exception occurs, VSR[VRT+32] and FPRF are not modified, and FR and FI are set to 0.

See Table 7.10, "VSX Scalar Floating-Point Final Result," on page 618.

Special Registers Altered:

Register	*Field(s)*
FPSCR	FPRF FR FI FX VXSNAN VXISI OX UX XX

VSR Data Layout for xsaddqp[o]



[17] The quad-precision default Quiet NaN is the value, 0x7FFF_8000_0000_0000_0000_0000_0000.

src1 \ src2	−Infinity	−NZF	−Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
−Infinity	v ← −Infinity					v ← dQNaN vxisi_flag ← 1	v ← src2	v ← quiet(src2) vxsnan_flag ← 1
−NZF		v ← add(src1,src2)	v ← src1		v ← add(src1,src2)			
−Zero		v ← src2	v ← −Zero	v ← Rezd	v ← src2			
+Zero			v ← Rezd	v ← +Zero				
+NZF		v ← add(src1,src2)	v ← src1		v ← add(src1,src2)			
+Infinity	v ← dQNaN vxisi_flag ← 1	v ← +Infinity						
QNaN	v ← src1							v ← src1 vxsnan_flag ← 1
SNaN	v ← quiet(src1) vxsnan_flag ← 1							

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_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[2] sum, having unbounded significand precision and exponent range. When x = -y, v is considered to be an exact-zero-difference result (Rezd).
quiet(x)	Convert x to the corresponding Quiet NaN by setting the most significant fraction bit to 1.
v	The intermediate result having unbounded significand precision and unbounded exponent range.

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.

Table 7.11: Actions for xsaddqp[o]

VSX Scalar Add Single-Precision XX3-form

xsaddsp XT,XA,XB

60	T	A	B	0	AX	BX	TX
0	6	11	16	21	29	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

reset_xflags()

src1     ← bfp_CONVERT_FROM_BFP64(VSR[VRA+32].dword[0])
src2     ← bfp_CONVERT_FROM_BFP64(VSR[VRB+32].dword[0])
v        ← bfp_ADD(src1, src2)
rnd      ← bfp_ROUND_TO_BFP32(FPSCR.RN, v)
result32 ← bfp32_CONVERT_FROM_BFP(rnd)
result64 ← bfp64_CONVERT_FROM_BFP(rnd)

if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
if vxisi_flag=1 then SetFX(FPSCR.VXISI)
if ox_flag=1 then SetFX(FPSCR.OX)
if ux_flag=1 then SetFX(FPSCR.UX)
if xx_flag=1 then SetFX(FPSCR.XX)

vx_flag  ← vxsnan_flag | vxisi_flag
vex_flag ← FPSCR.VE & vx_flag

if vex_flag=0 then do
   VSR[32×TX+T].dword[0] ← result64
   VSR[32×TX+T].dword[1] ← 0x0000_0000_0000_0000
   FPSCR.FPRF ← fprf_CLASS_BFP32(result32)
   FPSCR.FR   ← inc_flag
   FPSCR.FI   ← xx_flag
end
else do
   FPSCR.FR   ← 0b0
   FPSCR.FI   ← 0b0
end
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

Let src1 be the double-precision floating-point value in doubleword element 0 of VSR[XA].

Let src2 be the double-precision floating-point value in doubleword element 0 of VSR[XB].

src2 is added[18] to src1, producing a sum having unbounded range and precision.

The sum is normalized[19].

See Table 7.12, “Actions for xsaddsp,” on page 623.

The intermediate result is rounded to single-precision using the rounding mode specified by RN.

See Table 7.9, “Scalar Floating-Point Intermediate Result Handling,” on page 617.

The result is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are set to 0.

FPRF is set to the class and sign of the result as represented in single-precision format. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.

If a trap-enabled invalid operation exception occurs, VSR[XT] and FPRF are not modified, and FR and FI are set to 0.

See Table 7.10, “VSX Scalar Floating-Point Final Result,” on page 618.

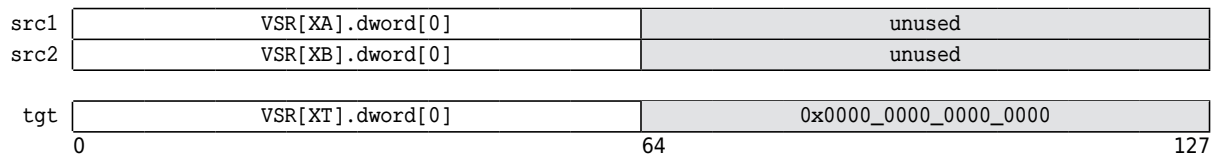
Special Registers Altered:

Register	*Field(s)*
FPSCR	FPRF FR FI FX OX UX XX VXSNAN VXISI

Programming Note

Previous versions of the architecture allowed the contents of doubleword 1 of the result register to be undefined. However, all processors that support this instruction write 0s into doubleword 1 of the result register, as is required by this version of the architecture.

VSR Data Layout for xsaddsp



[18] 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.

[19] 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.

src1 \ src2	−Infinity	−NZF	−Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
−Infinity	v ← −Infinity	v ← −Infinity	v ← −Infinity	v ← −Infinity	v ← −Infinity	v ← dQNaN vxisi_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
−NZF	v ← −Infinity	v ← A(src1,src2)	v ← src1	v ← src1	v ← A(src1,src2)	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
−Zero	v ← −Infinity	v ← src2	v ← −Zero	v ← Rezd	v ← src2	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Zero	v ← −Infinity	v ← src2	v ← Rezd	v ← +Zero	v ← src2	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+NZF	v ← −Infinity	v ← A(src1,src2)	v ← src1	v ← src1	v ← A(src1,src2)	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Infinity	v ← dQNaN vxisi_flag ← 1	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
QNaN	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1 vxsnan_flag ← 1
SNaN	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

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 signficand precision and unbounded exponent range.

Table 7.12: Actions for xsaddsp

VSX Scalar Divide Double-Precision XX3-form

xsdivdp XT,XA,XB

60	T	A	B	56	AX	BX	TX
0	6	11	16	21	29	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

reset_xflags()

src1   ← bfp_CONVERT_FROM_BFP64(VSR[32×AX+A].dword[0])
src2   ← bfp_CONVERT_FROM_BFP64(VSR[32×BX+B].dword[0])
v      ← bfp_DIVIDE(src1,src2)
rnd    ← bfp_ROUND_TO_BFP64(0b0,FPSCR.RN,v)
result ← bfp64_CONVERT_FROM_BFP(rnd)

if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
if vxidi_flag=1 then SetFX(FPSCR.VXIDI)
if vxzdz_flag=1 then SetFX(FPSCR.VXZDZ)
if ox_flag=1 then SetFX(FPSCR.OX)
if ux_flag=1 then SetFX(FPSCR.UX)
if xx_flag=1 then SetFX(FPSCR.XX)
if zx_flag=1 then SetFX(FPSCR.ZX)

vx_flag  ← vxsnan_flag | vxidi_flag | vxzdz_flag
vex_flag ← FPSCR.VE & vx_flag
zex_flag ← FPSCR.ZE & zx_flag

if vex_flag=0 & zex_flag=0 then do
   VSR[32×TX+T].dword[0] ← result
   VSR[32×TX+T].dword[1] ← 0x0000_0000_0000_0000
   FPSCR.FPRF ← fprf_CLASS_BFP64(result)
   FPSCR.FR   ← inc_flag
   FPSCR.FI   ← xx_flag
end
else do
   FPSCR.FR   ← 0b0
   FPSCR.FI   ← 0b0
end
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

Let src1 be the double-precision floating-point value in doubleword element 0 of VSR[XA].

Let src2 be the double-precision floating-point value in doubleword element 0 of VSR[XB].

src1 is divided[20] by src2, producing a quotient having unbounded range and precision.

The quotient is normalized[21].

See *Actions for xsdivdp* (p. 625).

The intermediate result is rounded to double-precision using the rounding mode specified by RN.

See Table 7.9, “Scalar Floating-Point Intermediate Result Handling,” on page 617.

The result is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are set to 0.

FPRF is set to the class and sign of the result. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.

If a trap-enabled invalid operation exception or a trap-enabled zero divide exception occurs, VSR[XT] and FPRF are not modified, and FR and FI are set to 0.

See Table 7.10, “VSX Scalar Floating-Point Final Result,” on page 618.

Special Registers Altered:

Register	*Field(s)*
FPSCR	FPRF FR FI FX OX UX ZX XX VXSNAN VXIDI VXZDZ

VSR Data Layout for xsdivdp

[20] Floating-point division is based on exponent subtraction and division of the significands.
[21] 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.

		src2							
		−Infinity	**−NZF**	**−Zero**	**+Zero**	**+NZF**	**+Infinity**	**QNaN**	**SNaN**
src1	**−Infinity**	v ← dQNaN vxidi_flag ← 1	v ← +Infinity	v ← +Infinity	v ← −Infinity	v ← −Infinity	v ← dQNaN vxidi_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	−NZF	v ← +Zero	v ← D(src1,src2)	v ← +Infinity zx_flag ← 1	v ← −Infinity zx_flag ← 1	v ← D(src1,src2)	v ← −Zero	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	−Zero	v ← +Zero	v ← +Zero	v ← dQNaN vxzdz_flag ← 1	v ← dQNaN vxzdz_flag ← 1	v ← −Zero	v ← −Zero	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+Zero	v ← −Zero	v ← −Zero	v ← dQNaN vxzdz_flag ← 1	v ← dQNaN vxzdz_flag ← 1	v ← +Zero	v ← +Zero	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+NZF	v ← −Zero	v ← D(src1,src2)	v ← −Infinity zx_flag ← 1	v ← +Infinity zx_flag ← 1	v ← D(src1,src2)	v ← +Zero	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+Infinity	v ← dQNaN vxidi_flag ← 1	v ← −Infinity	v ← −Infinity	v ← +Infinity	v ← +Infinity	v ← dQNaN vxidi_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	QNaN	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1 vxsnan_flag ← 1
	SNaN	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

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 signficand precision and unbounded exponent range.

Table 7.13: Actions for xsdivdp

VSX Scalar Divide Quad-Precision [using round to Odd] X-form

xsdivqp	VRT,VRA,VRB	(RO=0)
xsdivqpo	VRT,VRA,VRB	(RO=1)

63	VRT	VRA	VRB	548	RO
0	6	11	16	21	31

```
if MSR.VSX=0 then VSX_Unavailable()

reset_xflags()

src1   ← bfp_CONVERT_FROM_BFP128(VSR[VRA+32])
src2   ← bfp_CONVERT_FROM_BFP128(VSR[VRB+32])
v      ← bfp_DIVIDE(src1, src2)
rnd    ← bfp_ROUND_TO_BFP128(RO, FPSCR.RN, v)
result ← bfp128_CONVERT_FROM_BFP(rnd)

if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
if vxidi_flag=1 then SetFX(FPSCR.VXIDI)
if vxzdz_flag=1 then SetFX(FPSCR.VXZDZ)
if ox_flag=1 then SetFX(FPSCR.OX)
if ux_flag=1 then SetFX(FPSCR.UX)
if zx_flag=1 then SetFX(FPSCR.ZX)
if xx_flag=1 then SetFX(FPSCR.XX)

vx_flag ← vxsnan_flag | vxidi_flag | vxzdz_flag
vex_flag ← FPSCR.VE & vx_flag
zex_flag ← FPSCR.ZE & zx_flag

if vex_flag=0 & zex_flag=0 then do
   VSR[VRT+32] ← result
   FPSCR.FPRF  ← fprf_CLASS_BFP128(result)
end
FPSCR.FR ← (vx_flag=0) & (zx_flag=0) & inc_flag
FPSCR.FI ← (vx_flag=0) & (zx_flag=0) & xx_flag
```

Let src1 be the floating-point value in VSR[VRA+32] represented in quad-precision format.

Let src2 be the floating-point value in VSR[VRB+32] represented in quad-precision format.

If either src1 or src2 is a Signalling NaN, an Invalid Operation exception occurs and VXSNAN is set to 1

If src1 and src2 are Infinity values, an Invalid Operation exception occurs and VXIDI is set to 1.

If src1 and src2 are Zero values, an Invalid Operation exception occurs and VXZDZ is set to 1.

If src1 is a finite value and src2 is a Zero value, an Zero Divide exception occurs and ZX is set to 1.

If src1 is a Signalling NaN, the result is the Quiet NaN corresponding to src1.

Otherwise, if src1 is a Quiet NaN, the result is src1.

Otherwise, if src2 is a Signalling NaN, the result is the Quiet NaN corresponding to src2.

Otherwise, if src2 is a Quiet NaN, the result is src2.

Otherwise, if src1 and src2 are Infinity values, or if src1 and src2 are Zero values, the result is the default Quiet NaN[22].

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 7.14, “Actions for xsdivqp[o],” on page 627.
>
> 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 7.9, “Scalar Floating-Point Intermediate Result Handling,” on page 617.

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.

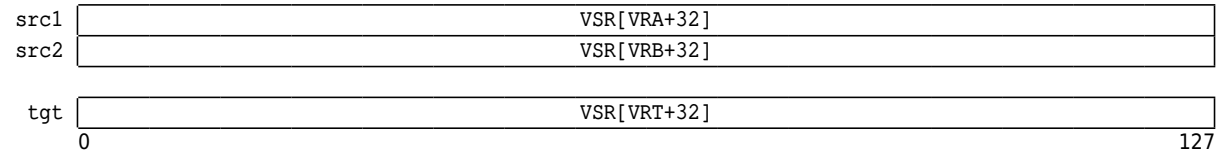
See Table 7.10, “VSX Scalar Floating-Point Final Result,” on page 618.

Special Registers Altered:

Register	*Field(s)*
FPSCR	FPRF FR FI FX VXSNAN VXIDI VXZDZ OX UX ZX XX

[22] The quad-precision default Quiet NaN is the value, 0x7FFF_8000_0000_0000_0000_0000_0000.

VSR Data Layout for xsdivqp[o]



src1 \ src2	−Infinity	−NZF	−Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
−Infinity	v ← dQNaN vxidi_flag ← 1	v ← +Infinity		v ← −Infinity		v ← dQNaN vxidi_flag ← 1	v ← src2	v ← quiet(src2) vxsnan_flag ← 1
−NZF	v ← +Zero	v ← Div(src1,src2)	v ← +Infinity zx_flag ← 1	v ← −Infinity zx_flag ← 1	v ← Div(src1,src2)	v ← −Zero		
−Zero			v ← dQNaN vxzdz_flag ← 1					
+Zero	v ← −Zero					v ← +Zero		
+NZF		v ← Div(src1,src2)	v ← −Infinity zx_flag ← 1	v ← +Infinity zx_flag ← 1	v ← Div(src1,src2)			
+Infinity	v ← dQNaN vxidi_flag ← 1	v ← −Infinity		v ← +Infinity		v ← dQNaN vxidi_flag ← 1		
QNaN	v ← src1							v ← src1 vxsnan_flag ← 1
SNaN	v ← quiet(src1) vxsnan_flag ← 1							

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.
Div(x,y)	The floating-point value x is divided by floating-point value y. Return the normalized[23] 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 7.14: Actions for xsdivqp[o]

[23] Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.

VSX Scalar Divide Single-Precision XX3-form

xsdivsp XT,XA,XB

60	T	A	B	24	AX	BX	TX
0	6	11	16	21	29	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

reset_xflags()

src1 ← bfp_CONVERT_FROM_BFP64(VSR[32×AX+A].dword[0])
src2 ← bfp_CONVERT_FROM_BFP64(VSR[32×BX+B].dword[0])
v    ← bfp_DIVIDE(src1,src2)

rnd      ← bfp_ROUND_TO_BFP32(FPSCR.RN,v)
result32 ← bfp32_CONVERT_FROM_BFP(rnd)
result64 ← bfp64_CONVERT_FROM_BFP(rnd)

if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
if vxidi_flag=1 then SetFX(FPSCR.VXIDI)
if vxzdz_flag=1 then SetFX(FPSCR.VXZDZ)
if ox_flag=1 then SetFX(FPSCR.OX)
if ux_flag=1 then SetFX(FPSCR.UX)
if xx_flag=1 then SetFX(FPSCR.XX)
if zx_flag=1 then SetFX(FPSCR.ZX)

vx_flag  ← vxsnan_flag | vxidi_flag | vxzdz_flag

vex_flag ← FPSCR.VE & vx_flag
zex_flag ← FPSCR.ZE & zx_flag

if vex_flag=0 & zex_flag=0 then do
   VSR[32×TX+T].dword[0] ← result64
   VSR[32×TX+T].dword[1] ← 0x0000_0000_0000_0000
   FPSCR.FPRF ← fprf_CLASS_BFP32(result32)
   FPSCR.FR   ← inc_flag
   FPSCR.FI   ← xx_flag
end
else do
   FPSCR.FR   ← 0b0
   FPSCR.FI   ← 0b0
end
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

Let src1 be the double-precision floating-point value in doubleword element 0 of VSR[XA].

Let src2 be the double-precision floating-point value in doubleword element 0 of VSR[XB].

src1 is divided[24] by src2, producing a quotient having unbounded range and precision.

The quotient is normalized[25].

See Table 7.15, “Actions for xsdivsp,” on page 629.

The intermediate result is rounded to single-precision using the rounding mode specified by RN.

See Table 7.9, “Scalar Floating-Point Intermediate Result Handling,” on page 617.

The result is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are set to 0.

FPRF is set to the class and sign of the result as represented in single-precision format. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.

If a trap-enabled invalid operation exception or a trap-enabled zero divide exception occurs, VSR[XT] and FPRF are not modified, and FR and FI are set to 0.

See Table 7.10, “VSX Scalar Floating-Point Final Result,” on page 618.

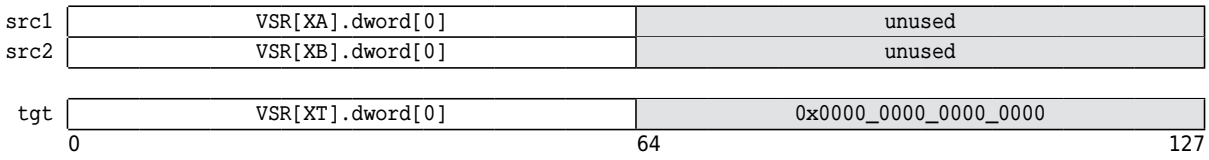
Special Registers Altered:

Register	*Field(s)*
FPSCR	FPRF FR FI FX OX UX ZX XX VXSNAN VXIDI VXZDZ

Programming Note

Previous versions of the architecture allowed the contents of doubleword 1 of the result register to be undefined. However, all processors that support this instruction write 0s into doubleword 1 of the result register, as is required by this version of the architecture.

VSR Data Layout for xsdivsp



[24] Floating-point division is based on exponent subtraction and division of the significands.
[25] 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.

		src2							
		−Infinity	**−NZF**	**−Zero**	**+Zero**	**+NZF**	**+Infinity**	**QNaN**	**SNaN**
src1	**−Infinity**	v ← dQNaN vxidi_flag ← 1	v ← +Infinity	v ← +Infinity	v ← −Infinity	v ← −Infinity	v ← dQNaN vxidi_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	−NZF	v ← +Zero	v ← D(src1,src2)	v ← +Infinity zx_flag ← 1	v ← −Infinity zx_flag ← 1	v ← D(src1,src2)	v ← −Zero	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	−Zero	v ← +Zero	v ← +Zero	v ← dQNaN vxzdz_flag ← 1	v ← dQNaN vxzdz_flag ← 1	v ← −Zero	v ← −Zero	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+Zero	v ← −Zero	v ← −Zero	v ← dQNaN vxzdz_flag ← 1	v ← dQNaN vxzdz_flag ← 1	v ← +Zero	v ← +Zero	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+NZF	v ← −Zero	v ← D(src1,src2)	v ← −Infinity zx_flag ← 1	v ← +Infinity zx_flag ← 1	v ← D(src1,src2)	v ← +Zero	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+Infinity	v ← dQNaN vxidi_flag ← 1	v ← −Infinity	v ← −Infinity	v ← +Infinity	v ← +Infinity	v ← dQNaN vxidi_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	QNaN	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1 vxsnan_flag ← 1
	SNaN	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

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 signficand precision and unbounded exponent range.

Table 7.15: Actions for xsdivsp

VSX Scalar Multiply Double-Precision XX3-form

xsmuldp XT,XA,XB

60	T	A	B	48	AX	BX	TX
0	6	11	16	21	29	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

reset_xflags()

src1   ← bfp_CONVERT_FROM_BFP64(VSR[32×AX+A].dword[0])
src2   ← bfp_CONVERT_FROM_BFP64(VSR[32×BX+B].dword[0])
v      ← bfp_MULTIPLY(src1,src2)
rnd    ← bfp_ROUND_TO_BFP64(0b0,FPSCR.RN,v)
result ← bfp64_CONVERT_FROM_BFP(rnd)

if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
if vximz_flag=1 then SetFX(FPSCR.VXIMZ)
if ox_flag=1 then SetFX(FPSCR.OX)
if ux_flag=1 then SetFX(FPSCR.UX)
if xx_flag=1 then SetFX(FPSCR.XX)

vx_flag  ← vxsnan_flag | vximz_flag
vex_flag ← FPSCR.VE & vx_flag

if vex_flag=0 then do
   VSR[32×TX+T].dword[0] ← result
   VSR[32×TX+T].dword[1] ← 0x0000_0000_0000_0000
   FPSCR.FPRF ← fprf_CLASS_BFP64(result)
   FPSCR.FR   ← inc_flag
   FPSCR.FI   ← xx_flag
end
else do
   FPSCR.FR   ← 0b0
   FPSCR.FI   ← 0b0
end
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

Let src1 be the double-precision floating-point value in doubleword element 0 of VSR[XA].

Let src2 be the double-precision floating-point value in doubleword element 0 of VSR[XB].

src1 is multiplied[26] by src2, producing a product having unbounded range and precision.

The product is normalized[27].

See Table 7.16.

The intermediate result is rounded to double-precision using the rounding mode specified by RN.

See Table 7.9, “Scalar Floating-Point Intermediate Result Handling,” on page 617.

The result is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are set to 0.

FPRF is set to the class and sign of the result. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.

If a trap-enabled invalid operation exception occurs, VSR[XT] and FPRF are not modified, and FR and FI are set to 0.

See Table 7.10, “VSX Scalar Floating-Point Final Result,” on page 618.

Special Registers Altered:

Register	*Field(s)*
FPSCR	FPRF FR FI FX OX UX XX VXSNAN VXIMZ

Programming Note

Previous versions of the architecture allowed the contents of doubleword 1 of the result register to be undefined. However, all processors that support this instruction write 0s into doubleword 1 of the result register, as is required by this version of the architecture.

VSR Data Layout for xsmuldp

[26] Floating-point multiplication is based on exponent addition and multiplication of the significands.
[27] 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.

		src2							
		−Infinity	**−NZF**	**−Zero**	**+Zero**	**+NZF**	**+Infinity**	**QNaN**	**SNaN**
src1	**−Infinity**	v ← +Infinity	v ← +Infinity	v ← dQNaN vximz_flag ← 1	v ← dQNaN vximz_flag ← 1	v ← −Infinity	v ← −Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	−NZF	v ← +Infinity	v ← M(src1,src2)	v ← +Zero	v ← −Zero	v ← M(src1,src2)	v ← −Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	−Zero	v ← dQNaN vximz_flag ← 1	v ← +Zero	v ← +Zero	v ← −Zero	v ← −Zero	v ← dQNaN vximz_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+Zero	v ← dQNaN vximz_flag ← 1	v ← −Zero	v ← −Zero	v ← +Zero	v ← +Zero	v ← dQNaN vximz_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+NZF	v ← −Infinity	v ← M(src1,src2)	v ← −Zero	v ← +Zero	v ← M(src1,src2)	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+Infinity	v ← −Infinity	v ← −Infinity	v ← dQNaN vximz_flag ← 1	v ← dQNaN vximz_flag ← 1	v ← +Infinity	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	QNaN	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1 vxsnan_flag ← 1
	SNaN	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

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 signficand precision and unbounded exponent range.

Table 7.16: Actions for xsmuldp

VSX Scalar Multiply Quad-Precision [using round to Odd] X-form

xsmulqp	VRT,VRA,VRB	(RO=0)
xsmulqpo	VRT,VRA,VRB	(RO=1)

63	VRT	VRA	VRB	36	RO
0	6	11	16	21	31

```
if MSR.VSX=0 then VSX_Unavailable()

reset_xflags()

src1   ← bfp_CONVERT_FROM_BFP128(VSR[VRA+32])
src2   ← bfp_CONVERT_FROM_BFP128(VSR[VRB+32])
v      ← bfp_MULTIPLY(src1, src2)
rnd    ← bfp_ROUND_TO_BFP128(RO, FPSCR.RN, v)
result ← bfp128_CONVERT_FROM_BFP(rnd)

if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
if vximz_flag=1 then SetFX(FPSCR.VXIMZ)
if ox_flag=1 then SetFX(FPSCR.OX)
if ux_flag=1 then SetFX(FPSCR.UX)
if xx_flag=1 then SetFX(FPSCR.XX)

vx_flag  ← vxsnan_flag | vximz_flag
vex_flag ← FPSCR.VE & vx_flag

if vex_flag=0 then do
   VSR[VRT+32] ← result
   FPSCR.FPRF  ← fprf_CLASS_BFP128(result)
end
FPSCR.FR ← (vx_flag=0) & inc_flag
FPSCR.FI ← (vx_flag=0) & xx_flag
```

Let src1 be the floating-point value in VSR[VRA+32] represented in quad-precision format.

Let src2 be the floating-point value in VSR[VRB+32] represented in quad-precision format.

If either src1 or src2 is a Signalling NaN, an Invalid Operation exception occurs and VXSNAN is set to 1.

If src1 is an Infinity value and src2 is a Zero value, or if src1 is a Zero value and src2 is an Infinity value, an Invalid Operation exception occurs and VXIMZ is set to 1.

If src1 is a Signalling NaN, the result is the Quiet NaN corresponding to src1.

Otherwise, if src1 is a Quiet NaN, the result is src1.

Otherwise, if src2 is a Signalling NaN, the result is the Quiet NaN corresponding to src2.

Otherwise, if src2 is a Quiet NaN, the result is src2.

Otherwise, if src1 is an Infinity value and src2 is a Zero value, or if src1 is a Zero value and src2 is an Infinity value, the result is the default Quiet NaN[28].

Otherwise, do the following.

The normalized product of src1 multiplied by src2 is produced with unbounded significand precision and exponent range.

See Table 7.17. “Actions for xsmulqp[o]”.

If the intermediate result is *Tiny* (i.e., the unbiased exponent is less than −16382) and UE=0, the significand is shifted right N bits, where N is the difference between −16382 and the unbiased exponent of the intermediate result. The exponent of the intermediate result is set to the value −16382.

If RO=1, let the rounding mode be Round to Odd. Otherwise, let the rounding mode be specified by RN. Unless the result is an Infinity or a Zero, the intermediate result is rounded to quad-precision using the specified rounding mode.

See Table 7.9, “Scalar Floating-Point Intermediate Result Handling,” on page 617.

The result is placed into VSR[VRT+32] in quad-precision format.

FPRF is set to the class and sign of the result. FR is set to indicate if the rounded result was incremented. FI is set to indicate the result is inexact.

If a trap-disabled Invalid Operation exception occurs, FR and FI are set to 0.

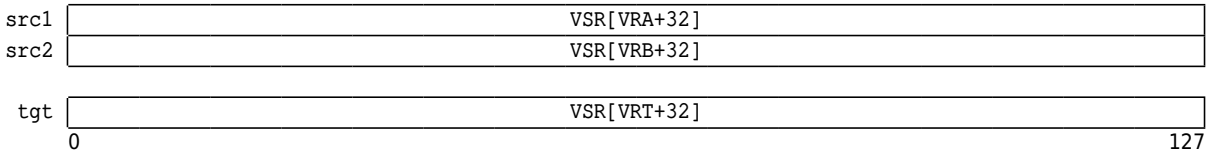
If a trap-enabled Invalid Operation exception occurs, VSR[VRT+32] and FPRF are not modified, and FR and FI are set to 0.

See Table 7.10, “VSX Scalar Floating-Point Final Result,” on page 618.

Special Registers Altered:

Register	*Field(s)*
FPSCR	FPRF FR FI FX VXSNAN VXIMZ OX UX XX

VSR Data Layout for xsmulqp[o]



[28] The quad-precision default Quiet NaN is the value, 0x7FFF_8000_0000_0000_0000_0000.

src1 \ src2	−Infinity	−NZF	−Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
−Infinity	v ← +Infinity		v ← dQNaN vximz_flag ← 1			v ← −Infinity	v ← src2	v ← quiet(src2) vxsnan_flag ← 1
−NZF		v ← mul(src1,src2)			v ← mul(src1,src2)			
−Zero	v ← dQNaN vximz_flag ← 1		v ← +Zero	v ← −Zero		v ← dQNaN vximz_flag ← 1		
+Zero			v ← −Zero	v ← +Zero				
+NZF		v ← mul(src1,src2)			v ← mul(src1,src2)			
+Infinity	v ← −Infinity		v ← dQNaN vximz_flag ← 1			v ← +Infinity		
QNaN	v ← src1							v ← src1 vxsnan_flag ← 1
SNaN	v ← quiet(src1) vxsnan_flag ← 1							

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 by the floating-point value y. (Floating-point multiplication is based on exponent addition and multiplication of the significands.) 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.

Table 7.17: Actions for xsmulqp[o]

VSX Scalar Multiply Single-Precision XX3-form

xsmulsp XT,XA,XB

60	T	A	B	16	AX	BX	TX
0	6	11	16	21	29	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

reset_xflags()

src1 ← bfp_CONVERT_FROM_BFP64(VSR[32×AX+A].dword[0])
src2 ← bfp_CONVERT_FROM_BFP64(VSR[32×BX+B].dword[0])
v    ← bfp_MULTIPLY(src1,src2)

rnd      ← bfp_ROUND_TO_BFP32(FPSCR.RN,v)
result32 ← bfp32_CONVERT_FROM_BFP(rnd)
result64 ← bfp64_CONVERT_FROM_BFP(rnd)

if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
if vximz_flag=1 then SetFX(FPSCR.VXIMZ)
if ox_flag=1 then SetFX(FPSCR.OX)
if ux_flag=1 then SetFX(FPSCR.UX)
if xx_flag=1 then SetFX(FPSCR.XX)

vx_flag  ← vxsnan_flag | vximz_flag
vex_flag ← FPSCR.VE & vx_flag

if vex_flag=0 then do
   VSR[32×TX+T].dword[0] ← result64
   VSR[32×TX+T].dword[1] ← 0x0000_0000_0000_0000
   FPSCR.FPRF ← fprf_CLASS_BFP32(result32)
   FPSCR.FR   ← inc_flag
   FPSCR.FI   ← xx_flag
end
else do
   FPSCR.FR   ← 0b0
   FPSCR.FI   ← 0b0
end
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

Let src1 be the double-precision floating-point value in doubleword element 0 of VSR[XA].

Let src2 be the double-precision floating-point value in doubleword element 0 of VSR[XB].

src1 is multiplied[29] by src2, producing a product having unbounded range and precision.

The product is normalized[30].

See Table 7.18, “Actions for xsmulsp,” on page 635.

The intermediate result is rounded to single-precision using the rounding mode specified by RN.

See Table 7.9, “Scalar Floating-Point Intermediate Result Handling,” on page 617.

The result is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are set to 0.

FPRF is set to the class and sign of the result as represented in single-precision format. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.

If a trap-enabled invalid operation exception occurs, VSR[XT] and FPRF are not modified, and FR and FI are set to 0.

See Table 7.10, “VSX Scalar Floating-Point Final Result,” on page 618.

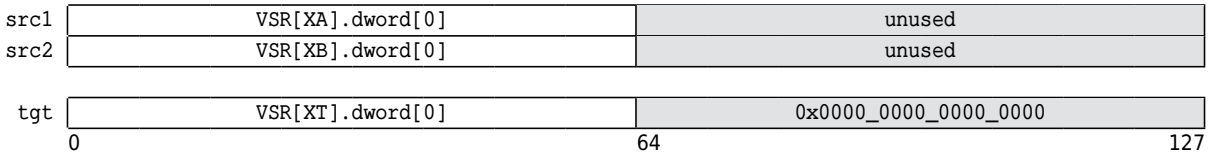
Special Registers Altered:

Register	*Field(s)*
FPSCR	FPRF FR FI FX OX UX XX VXSNAN VXIMZ

Programming Note

Previous versions of the architecture allowed the contents of doubleword 1 of the result register to be undefined. However, all processors that support this instruction write 0s into doubleword 1 of the result register, as is required by this version of the architecture.

VSR Data Layout for xsmulsp



[29] Floating-point multiplication is based on exponent addition and multiplication of the significands.
[30] 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.

		src2							
		−Infinity	**−NZF**	**−Zero**	**+Zero**	**+NZF**	**+Infinity**	**QNaN**	**SNaN**
src1	**−Infinity**	v ← +Infinity	v ← +Infinity	v ← dQNaN vximz_flag ← 1	v ← dQNaN vximz_flag ← 1	v ← −Infinity	v ← −Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	−NZF	v ← +Infinity	v ← M(src1,src2)	v ← +Zero	v ← −Zero	v ← M(src1,src2)	v ← −Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	−Zero	v ← dQNaN vximz_flag ← 1	v ← +Zero	v ← +Zero	v ← −Zero	v ← −Zero	v ← dQNaN vximz_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+Zero	v ← dQNaN vximz_flag ← 1	v ← −Zero	v ← −Zero	v ← +Zero	v ← +Zero	v ← dQNaN vximz_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+NZF	v ← −Infinity	v ← M(src1,src2)	v ← −Zero	v ← +Zero	v ← M(src1,src2)	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+Infinity	v ← −Infinity	v ← −Infinity	v ← dQNaN vximz_flag ← 1	v ← dQNaN vximz_flag ← 1	v ← +Infinity	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	QNaN	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1 vxsnan_flag ← 1
	SNaN	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

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 signficand precision and unbounded exponent range.

Table 7.18: Actions for xsmulsp

VSX Scalar Square Root Double-Precision XX2-form

xssqrtdp XT,XB

60	T	///	B	75	BX	TX
0	6	11	16	21	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

reset_xflags()

src    ← bfp_CONVERT_FROM_BFP64(VSR[32×BX+B].dword[0])
v      ← bfp_SQUARE_ROOT(src)
rnd    ← bfp_ROUND_TO_BFP64(0b0,FPSCR.RN,v)
result ← bfp64_CONVERT_FROM_BFP(rnd)

if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
if vxsqrt_flag=1 then SetFX(FPSCR.VXSQRT)
if xx_flag=1     then SetFX(FPSCR.XX)

vx_flag  ← vxsnan_flag | vxsqrt_flag
vex_flag ← FPSCR.VE & vx_flag

if vex_flag=0 then do
   VSR[32×TX+T].dword[0] ← result
   VSR[32×TX+T].dword[1] ← 0x0000_0000_0000_0000
   FPSCR.FPRF ← fprf_CLASS_BFP64(result)
   FPSCR.FR   ← inc_flag
   FPSCR.FI   ← xx_flag
end
else do
   FPSCR.FR   ← 0b0
   FPSCR.FI   ← 0b0
end
```

Let XT be the value 32×TX + T.
Let 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.

See Table 7.19.

The intermediate result is rounded to double-precision using the rounding mode specified by RN.

See Table 7.9, “Scalar Floating-Point Intermediate Result Handling,” on page 617.

The result is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are set to 0.

FPRF is set to the class and sign of the result. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.

If a trap-enabled invalid operation exception occurs, VSR[XT] and FPRF are not modified, and FR and FI are set to 0.

See Table 7.10, “VSX Scalar Floating-Point Final Result,” on page 618.

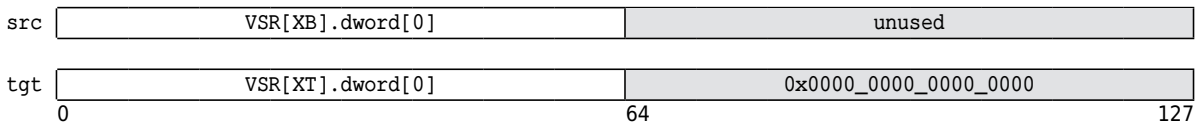
Special Registers Altered:

Register	*Field(s)*
FPSCR	FPRF FR FI FX XX
	VXSNAN VXSQRT

Programming Note

Previous versions of the architecture allowed the contents of doubleword 1 of the result register to be undefined. However, all processors that support this instruction write 0s into doubleword 1 of the result register, as is required by this version of the architecture.

VSR Data Layout for xssqrtdp



src							
−Infinity	**−NZF**	**−Zero**	**+Zero**	**+NZF**	**+Infinity**	**QNaN**	**SNaN**
v ← dQNaN vxsqrt_flag ← 1	v ← dQNaN vxsqrt_flag ← 1	v ← +Zero	v ← +Zero	v ← SQRT(src)	v ← +Infinity	v ← src	v ← Q(src) 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 7.19: Actions for xssqrtdp

VSX Scalar Square Root Quad-Precision [using round to Odd] X-form

xssqrtqp	VRT,VRB	(RO=0)
xssqrtqpo	VRT,VRB	(RO=1)

63	VRT	27	VRB	804	RO
0	6	11	16	21	31

```
if MSR.VSX=0 then VSX_Unavailable()

reset_xflags()

src    ← bfp_CONVERT_FROM_BFP128(VSR[VRB+32])
v      ← bfp_SQUARE_ROOT(src)
rnd    ← bfp_ROUND_TO_BFP128(RO,FPSCR.RN,v)
result ← bfp128_CONVERT_FROM_BFP(rnd)

if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
if vxsqrt_flag=1 then SetFX(FPSCR.VXSQRT)
if xx_flag=1     then SetFX(FPSCR.XX)

vx_flag  ← vxsnan_flag | vxsqrt_flag
vex_flag ← FPSCR.VE & vx_flag

if vex_flag=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 src be the floating-point value in VSR[VRB+32] represented in quad-precision format.

If src is a Signalling NaN, an Invalid Operation exception occurs and VXSNAN is set to 1.

If src is a negative, non-zero value, an Invalid Operation exception occurs and VXSQRT is set to 1.

If src is a Signalling NaN, the result is the Quiet NaN corresponding to src.

Otherwise, if src is a Quiet NaN, the result is src.

Otherwise, if src is a negative value, the result is the default Quiet NaN[31].

Otherwise, do the following.

The normalized square root of src is produced with unbounded significand precision and exponent range.

See Table 7.20, “Actions for xssqrtqp[o],” on page 639.

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 499 for a description of rounding modes.

If there is loss of precision, an Inexact exception occurs.

See Table 7.9, “Scalar Floating-Point Intermediate Result Handling,” on page 617.

The result is placed into VSR[VRT+32] in quad-precision format.

FPRF is set to the class and sign of the result. FR is set to indicate if the rounded result was incremented. FI is set to indicate the result is inexact.

If a trap-disabled Invalid Operation exception occurs, FPRF is set to an undefined value, and FR and FI are set to 0.

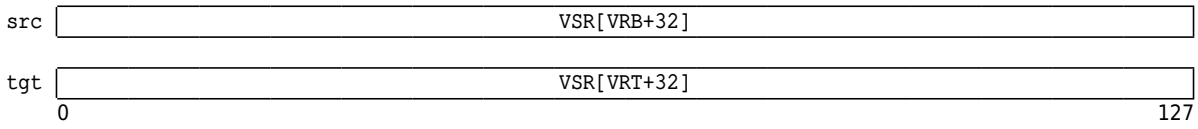
If a trap-enabled Invalid Operation exception occurs, VSR[VRT+32] and FPRF are not modified, and FR and FI are set to 0.

See Table 7.10, “VSX Scalar Floating-Point Final Result,” on page 618.

Special Registers Altered:

Register	*Field(s)*
FPSCR	FPRF FR FI FX VXSNAN VXSQRT XX

VSR Data Layout for xssqrtqp[o]



[31] The quad-precision default Quiet NaN is the value, 0x7FFF_8000_0000_0000_0000_0000_0000.

src							
−Infinity	**−NZF**	**−Zero**	**+Zero**	**+NZF**	**+Infinity**	**QNaN**	**SNaN**
v ← dQNaN vxsqrt_flag ← 1	v ← dQNaN vxsqrt_flag ← 1	v ← +Zero	v ← +Zero	v ← sqrt(src)	v ← +Infinity	v ← src	v ← quiet(src) vxsnan_flag ← 1

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 square root of floating-point value `x`, having unbounded significand precision and exponent range. (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.)
quiet(x)	Convert x to the corresponding Quiet NaN.
v	The intermediate result having unbounded significand precision and unbounded exponent range.

Table 7.20: Actions for xssqrtqp[o]

VSX Scalar Square Root Single-Precision XX2-form

xssqrtsp XT,XB

60	T	///	B	11	BX	TX
0	6	11	16	21	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

reset_xflags()

src ← bfp_CONVERT_FROM_BFP64(VSR[32×BX+B].dword[0])
v   ← bfp_SQUARE_ROOT(src)

rnd      ← bfp_ROUND_TO_BFP32(FPSCR.RN,v)
result32 ← bfp32_CONVERT_FROM_BFP(rnd)
result64 ← bfp64_CONVERT_FROM_BFP(rnd)

if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
if vxsqrt_flag=1 then SetFX(FPSCR.VXSQRT)
if ox_flag=1 then SetFX(FPSCR.OX)
if ux_flag=1 then SetFX(FPSCR.UX)
if xx_flag=1 then SetFX(FPSCR.XX)

vx_flag ← vxsnan_flag | vxsqrt_flag

vex_flag ← FPSCR.VE & vx_flag

if vex_flag=0 then do
   VSR[32×TX+T].dword[0] ← result64
   VSR[32×TX+T].dword[1] ← 0x0000_0000_0000_0000
   FPSCR.FPRF ← fprf_CLASS_BFP32(result32)
   FPSCR.FR   ← inc_flag
   FPSCR.FI   ← xx_flag
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].

The unbounded-precision square root of src is produced.

See Table 7.19.

The intermediate result is rounded to single-precision using the rounding mode specified by RN.

See Table 7.9, “Scalar Floating-Point Intermediate Result Handling,” on page 617.

The result is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are set to 0.

FPRF is set to the class and sign of the result as represented in single-precision format. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.

If a trap-enabled invalid operation exception occurs, VSR[XT] and FPRF are not modified, and FR and FI are set to 0.

See Table 7.10, “VSX Scalar Floating-Point Final Result,” on page 618.

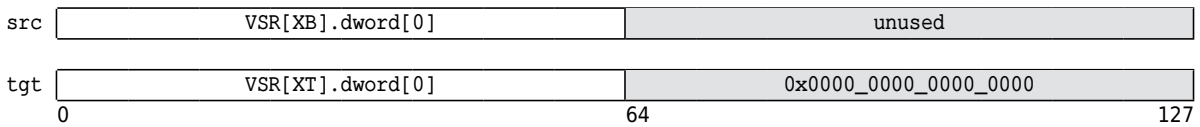
Special Registers Altered:

Register	*Field(s)*
FPSCR	FPRF FR FI FX OX UX XX VXSNAN VXSQRT

Programming Note

Previous versions of the architecture allowed the contents of doubleword 1 of the result register to be undefined. However, all processors that support this instruction write 0s into doubleword 1 of the result register, as is required by this version of the architecture.

VSR Data Layout for xssqrtsp



src							
−Infinity	**−NZF**	**−Zero**	**+Zero**	**+NZF**	**+Infinity**	**QNaN**	**SNaN**
v ← dQNaN vxsqrt_flag ← 1	v ← dQNaN vxsqrt_flag ← 1	v ← +Zero	v ← +Zero	v ← SQRT(src)	v ← +Infinity	v ← src	v ← Q(src) 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 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 7.21: Actions for xssqrtsp

VSX Scalar Subtract Double-Precision XX3-form

xssubdp XT,XA,XB

60	T	A	B	40	AX	BX	TX
0	6	11	16	21	29	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

reset_xflags()

src1   ← bfp_CONVERT_FROM_BFP64(VSR[32×AX+A].dword[0])
src2   ← bfp_CONVERT_FROM_BFP64(VSR[32×BX+B].dword[0])
v      ← bfp_ADD(src1,bfp_NEGATE(src2))
rnd    ← bfp_ROUND_TO_BFP64(0b0,FPSCR.RN,v)
result ← bfp64_CONVERT_FROM_BFP(rnd)

if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
if vxisi_flag=1 then SetFX(FPSCR.VXISI)
if ox_flag=1 then SetFX(FPSCR.OX)
if ux_flag=1 then SetFX(FPSCR.UX)
if xx_flag=1 then SetFX(FPSCR.XX)

vx_flag  ← vxsnan_flag | vxisi_flag
vex_flag ← FPSCR.VE & vx_flag

if vex_flag=0 then do
   VSR[32×TX+T].dword[0] ← result
   VSR[32×TX+T].dword[1] ← 0x0000_0000_0000_0000
   FPSCR.FPRF ← fprf_CLASS_BFP64(result)
   FPSCR.FR   ← inc_flag
   FPSCR.FI   ← xx_flag
end
else do
   FPSCR.FR   ← 0b0
   FPSCR.FI   ← 0b0
end
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

Let src1 be the double-precision floating-point value in doubleword element 0 of VSR[XA].

Let src2 be the double-precision floating-point value in doubleword element 0 of VSR[XB].

src2 is negated and added[32] to src1, producing a sum having unbounded range and precision.

See Table 7.22.

The sum is normalized[33].

The intermediate result is rounded to double-precision using the rounding mode specified by RN.

See Table 7.9, “Scalar Floating-Point Intermediate Result Handling,” on page 617.

The result is placed into doubleword element 0 of VSR[XT].

The contents of doubleword element 1 of VSR[XT] are set to 0.

FPRF is set to the class and sign of the result. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.

If a trap-enabled invalid operation exception occurs, VSR[XT] and FPRF are not modified, and FR and FI are set to 0.

See Table 7.10, “VSX Scalar Floating-Point Final Result,” on page 618.

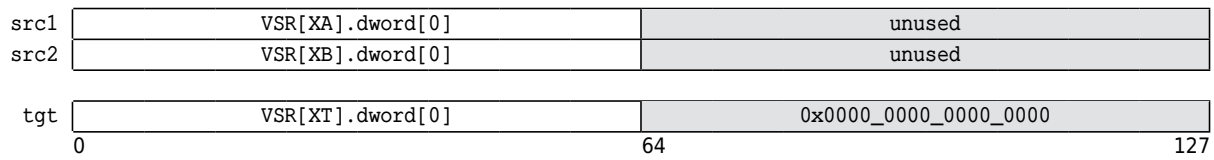
Special Registers Altered:

Register	*Field(s)*
FPSCR	FPRF FR FI FX OX UX XX VXSNAN VXISI

Programming Note

Previous versions of the architecture allowed the contents of doubleword 1 of the result register to be undefined. However, all processors that support this instruction write 0s into doubleword 1 of the result register, as is required by this version of the architecture.

VSR Data Layout for xssubdp



[32] 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.

[33] 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.

src1 \ src2	−Infinity	−NZF	−Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
−Infinity	v ← dQNaN vxisi_flag ← 1	v ← −Infinity	v ← −Infinity	v ← −Infinity	v ← −Infinity	v ← −Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
−NZF	v ← +Infinity	v ← S(src1,src2)	v ← src1	v ← src1	v ← S(src1,src2)	v ← −Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
−Zero	v ← +Infinity	v ← −src2	v ← Rezd	v ← −Zero	v ← −src2	v ← −Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Zero	v ← +Infinity	v ← −src2	v ← +Zero	v ← Rezd	v ← −src2	v ← −Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+NZF	v ← +Infinity	v ← S(src1,src2)	v ← src1	v ← src1	v ← S(src1,src2)	v ← −Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← dQNaN vxisi_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
QNaN	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1 vxsnan_flag ← 1
SNaN	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

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 signficand precision and unbounded exponent range.

Table 7.22: Actions for xssubdp

VSX Scalar Subtract Quad-Precision [using round to Odd] X-form

xssubqp	VRT,VRA,VRB	(RO=0)
xssubqpo	VRT,VRA,VRB	(RO=1)

63	VRT	VRA	VRB	516	RO
0	6	11	16	21	31

```
if MSR.VSX=0 then VSX_Unavailable()

reset_xflags()

src1   ← bfp_CONVERT_FROM_BFP128(VSR[VRA+32])
src2   ← bfp_CONVERT_FROM_BFP128(VSR[VRB+32])
v      ← bfp_ADD(src1, bfp_NEGATE(src2))
rnd    ← bfp_ROUND_TO_BFP128(RO,FPSCR.RN,v)
result ← bfp128_CONVERT_FROM_BFP(rnd)

if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
if vxisi_flag=1 then SetFX(FPSCR.VXISI)
if ox_flag=1 then SetFX(FPSCR.OX)
if ux_flag=1 then SetFX(FPSCR.UX)
if xx_flag=1 then SetFX(FPSCR.XX)

vx_flag  ← vxsnan_flag | vxisi_flag
vex_flag ← FPSCR.VE & vx_flag

if vex_flag=0 then do
   VSR[VRT+32] ← result
   FPSCR.FPRF  ← fprf_CLASS_BFP128(result)
end
FPSCR.FR ← (vx_flag=0) & inc_flag
FPSCR.FI ← (vx_flag=0) & xx_flag
```

Let src1 be the floating-point value in VSR[VRA+32] represented in quad-precision format.

Let src2 be the floating-point value in VSR[VRB+32] represented in quad-precision format.

If either src1 or src2 is a Signalling NaN, an Invalid Operation exception occurs and VXSNAN is set to 1.

If src1 and src2 are Infinity values having same signs, an Invalid Operation exception occurs and VXISI is set to 1.

If src1 is a Signalling NaN, the result is the Quiet NaN corresponding to src1.

Otherwise, if src1 is a Quiet NaN, the result is src1.

Otherwise, if src2 is a Signalling NaN, the result is the Quiet NaN corresponding to src2.

Otherwise, if src2 is a Quiet NaN, the result is src2.

Otherwise, if src1 and src2 are Infinity values having same signs, the result is the default Quiet NaN[34].

Otherwise, do the following.

The normalized sum of the negation of src2 added to src1 is produced with unbounded significand precision and exponent range.

See Table 7.23, "Actions for xssubqp[o]," on page 645.

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 7.9, "Scalar Floating-Point Intermediate Result Handling," on page 617.

The result is placed into VSR[VRT+32] in quad-precision format.

FPRF is set to the class and sign of the result. FR is set to indicate if the rounded result was incremented. FI is set to indicate the result is inexact.

If a trap-disabled Invalid Operation exception occurs, FPRF is set to an undefined value, and FR and FI are set to 0.

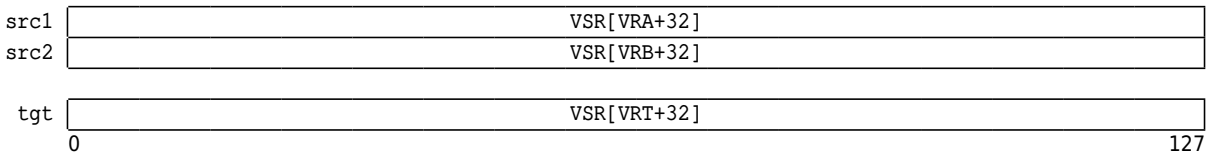
If a trap-enabled Invalid Operation exception occurs, VSR[VRT+32] and FPRF are not modified, and FR and FI are set to 0.

See Table 7.10, "VSX Scalar Floating-Point Final Result," on page 618.

Special Registers Altered:

Register	*Field(s)*
FPSCR	FPRF FR FI FX VXSNAN VXISI OX UX XX

VSR Data Layout for xssubqp[o]



[34] The quad-precision default Quiet NaN is the value, 0x7FFF_8000_0000_0000_0000_0000_0000.

src1 \ src2	−Infinity	−NZF	−Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
−Infinity	v ← dQNaN vxisi_flag ← 1					v ← −Infinity	v ← src2	v ← quiet(src2) vxsnan_flag ← 1
−NZF	v ← +Infinity	v ← sub(src1,src2)	v ← src1		v ← sub(src1,src2)			
−Zero		v ← −src2	v ← Rezd	v ← −Zero	v ← −src2			
+Zero			v ← +Zero	v ← Rezd				
+NZF		v ← sub(src1,src2)	v ← src1		v ← sub(src1,src2)			
+Infinity						v ← dQNaN vxisi_flag ← 1		
QNaN	v ← src1							v ← src1 vxsnan_flag ← 1
SNaN	v ← quiet(src1) vxsnan_flag ← 1							

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_0000`).
NZF	Nonzero finite number.
Rezd	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 (Rezd).
quiet(x)	Convert x to the corresponding Quiet NaN.
v	The intermediate result having unbounded significand precision and unbounded exponent range.

Table 7.23: Actions for xssubqp[o]

VSX Scalar Subtract Single-Precision XX3-form

xssubsp XT,XA,XB

60	T	A	B	8	AX	BX	TX
0	6	11	16	21	29	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

reset_xflags()

src1 ← bfp_CONVERT_FROM_BFP64(VSR[32×AX+A].dword[0])
src2 ← bfp_CONVERT_FROM_BFP64(VSR[32×BX+B].dword[0])
v    ← bfp_ADD(src1,bfp_NEGATE(src2))

rnd      ← bfp_ROUND_TO_BFP32(FPSCR.RN,v)
result32 ← bfp32_CONVERT_FROM_BFP(rnd)
result64 ← bfp64_CONVERT_FROM_BFP(rnd)

if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
if vxisi_flag=1 then SetFX(FPSCR.VXISI)
if ox_flag=1 then SetFX(FPSCR.OX)
if ux_flag=1 then SetFX(FPSCR.UX)
if xx_flag=1 then SetFX(FPSCR.XX)

vx_flag  ← vxsnan_flag | vxisi_flag
vex_flag ← FPSCR.VE & vx_flag

if vex_flag=0 then do
   VSR[32×TX+T].dword[0] ← result64
   VSR[32×TX+T].dword[1] ← 0x0000_0000_0000_0000
   FPSCR.FPRF ← fprf_CLASS_BFP32(result32)
   FPSCR.FR   ← inc_flag
   FPSCR.FI   ← xx_flag
end
else do
   FPSCR.FR   ← 0b0
   FPSCR.FI   ← 0b0
end
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

Let src1 be the double-precision floating-point value in doubleword element 0 of VSR[XA].

Let src2 be the double-precision floating-point value in doubleword element 0 of VSR[XB].

src2 is negated and added[35] to src1, producing the sum, v, having unbounded range and precision.

See Table 7.24, “Actions for xssubsp,” on page 647.

v is normalized[36] and rounded to single-precision using the rounding mode specified by RN.

See Table 7.9, “Scalar Floating-Point Intermediate Result Handling,” on page 617.

The result is placed into doubleword element 0 of VSR[XT].

The contents of doubleword element 1 of VSR[XT] are set to 0.

FPRF is set to the class and sign of the result as represented in single-precision format. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.

If a trap-enabled invalid operation exception occurs, VSR[XT] and FPRF are not modified, and FR and FI are set to 0.

See Table 7.10, “VSX Scalar Floating-Point Final Result,” on page 618.

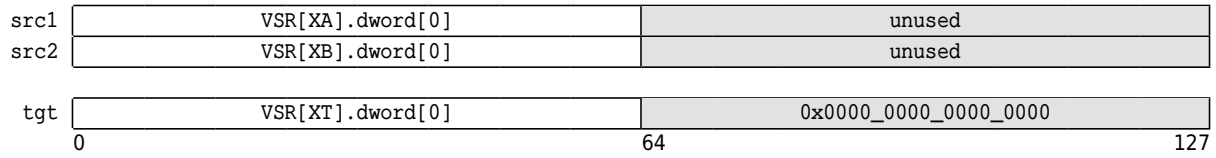
Special Registers Altered:

Register	*Field(s)*
FPSCR	FPRF FR FI FX OX UX XX VXSNAN VXISI

Programming Note

Previous versions of the architecture allowed the contents of doubleword 1 of the result register to be undefined. However, all processors that support this instruction write 0s into doubleword 1 of the result register, as is required by this version of the architecture.

VSR Data Layout for xssubsp



[35] 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.

[36] 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.

		src2							
		−Infinity	**−NZF**	**−Zero**	**+Zero**	**+NZF**	**+Infinity**	**QNaN**	**SNaN**
src1	**−Infinity**	v ← dQNaN vxisi_flag ← 1	v ← −Infinity	v ← −Infinity	v ← −Infinity	v ← −Infinity	v ← −Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	−NZF	v ← +Infinity	v ← S(src1,src2)	v ← src1	v ← src1	v ← S(src1,src2)	v ← −Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	−Zero	v ← +Infinity	v ← −src2	v ← Rezd	v ← −Zero	v ← −src2	v ← −Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+Zero	v ← +Infinity	v ← −src2	v ← +Zero	v ← Rezd	v ← −src2	v ← −Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+NZF	v ← +Infinity	v ← S(src1,src2)	v ← src1	v ← src1	v ← S(src1,src2)	v ← −Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← dQNaN vxisi_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	QNaN	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1 vxsnan_flag ← 1
	SNaN	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

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 signficand precision and unbounded exponent range.

Table 7.24: Actions for xssubsp

VSX Scalar Multiply-Add Type-A Double-Precision XX3-form

xsmaddadp XT,XA,XB

60	T	A	B	33	AX	BX	TX
0	6	11	16	21	29	30	31

VSX Scalar Multiply-Add Type-M Double-Precision XX3-form

xsmaddmdp XT,XA,XB

60	T	A	B	41	AX	BX	TX
0	6	11	16	21	29	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

reset_xflags()

if ''xsmaddadp'' then do
   src1 ← bfp_CONVERT_FROM_BFP64(VSR[32×AX+A].dword[0])
   src2 ← bfp_CONVERT_FROM_BFP64(VSR[32×TX+T].dword[0])
   src3 ← bfp_CONVERT_FROM_BFP64(VSR[32×BX+B].dword[0])
end
if ''xsmaddmdp'' then do
   src1 ← bfp_CONVERT_FROM_BFP64(VSR[32×AX+A].dword[0])
   src2 ← bfp_CONVERT_FROM_BFP64(VSR[32×BX+B].dword[0])
   src3 ← bfp_CONVERT_FROM_BFP64(VSR[32×TX+T].dword[0])
end

v      ← bfp_MULTIPLY_ADD(src1, src3, src2)
rnd    ← bfp_ROUND_TO_BFP64(0b0,FPSCR.RN, v)
result ← bfp64_CONVERT_FROM_BFP(rnd)

if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
if vximz_flag=1 then SetFX(FPSCR.VXIMZ)
if vxisi_flag=1 then SetFX(FPSCR.VXISI)
if ox_flag=1 then SetFX(FPSCR.OX)
if ux_flag=1 then SetFX(FPSCR.UX)
if xx_flag=1 then SetFX(FPSCR.XX)

vx_flag  ← vxsnan_flag | vximz_flag | vxisi_flag
vex_flag ← FPSCR.VE & vx_flag

if vex_flag=0 then do
   VSR[32×TX+T].dword[0] ← result
   VSR[32×TX+T].dword[1] ← 0x0000_0000_0000_0000
   FPSCR.FPRF ← fprf_CLASS_BFP64(result)
   FPSCR.FR   ← inc_flag
   FPSCR.FI   ← xx_flag
end
else do
   FPSCR.FR   ← 0b0
   FPSCR.FI   ← 0b0
end
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

Let src1 be the double-precision floating-point value in doubleword element 0 of VSR[XA].

For ***xsmaddadp***, do the following.

- Let src2 be the double-precision floating-point value in doubleword element 0 of VSR[XT].
- Let src3 be the double-precision floating-point value in doubleword element 0 of VSR[XB].

For ***xsmaddmdp***, 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[37] by src3, producing a product having unbounded range and precision.

See part 1 of Table 7.25.

src2 is added[38] to the product, producing a sum having unbounded range and precision.

The sum is normalized[39].

See part 2 of Table 7.25.

The intermediate result is rounded to double-precision using the rounding mode specified by RN.

See Table 7.9, “Scalar Floating-Point Intermediate Result Handling,” on page 617.

The result is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are set to 0.

FPRF is set to the class and sign of the result. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.

If a trap-enabled invalid operation exception occurs, VSR[XT] and FPRF are not modified, and FR and FI are set to 0.

See Table 7.10, “VSX Scalar Floating-Point Final Result,” on page 618.

[37] Floating-point multiplication is based on exponent addition and multiplication of the significands.

[38] 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.

[39] 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.

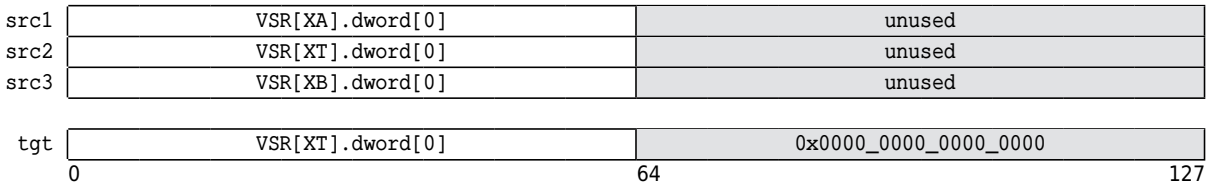
Special Registers Altered:

Register	*Field(s)*
FPSCR	FPRF FR FI FX OX UX XX VXSNAN VXISI VXIMZ

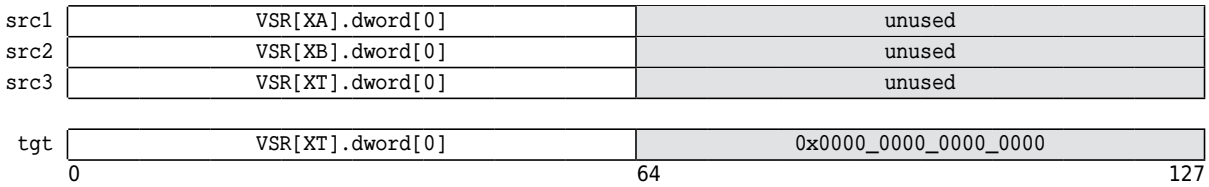
Programming Note

Previous versions of the architecture allowed the contents of doubleword 1 of the result register to be undefined. However, all processors that support this instruction write 0s into doubleword 1 of the result register, as is required by this version of the architecture.

VSR Data Layout for xsmaddadp



VSR Data Layout for xsmaddmdp



Part 1: Multiply	src3							
src1	−Infinity	−NZF	−Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
−Infinity	p ← +Infinity	p ← +Infinity	p ← dQNaN vximz_flag ← 1	p ← dQNaN vximz_flag ← 1	p ← −Infinity	p ← −Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
−NZF	p ← +Infinity	p ← M(src1,src3)	p ← +Zero	p ← −Zero	p ← M(src1,src3)	p ← −Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
−Zero	p ← dQNaN vximz_flag ← 1	p ← +Zero	p ← +Zero	p ← −Zero	p ← −Zero	p ← dQNaN vximz_flag ← 1	p ← src3	p ← Q(src3) vxsnan_flag ← 1
+Zero	p ← dQNaN vximz_flag ← 1	p ← −Zero	p ← −Zero	p ← +Zero	p ← +Zero	p ← dQNaN vximz_flag ← 1	p ← src3	p ← Q(src3) vxsnan_flag ← 1
+NZF	p ← −Infinity	p ← M(src1,src3)	p ← −Zero	p ← +Zero	p ← M(src1,src3)	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
+Infinity	p ← −Infinity	p ← −Infinity	p ← dQNaN vximz_flag ← 1	p ← dQNaN vximz_flag ← 1	p ← +Infinity	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
QNaN	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1 vxsnan_flag ← 1
SNaN	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1

Part 2: Add	src2							
p	−Infinity	−NZF	−Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
−Infinity	v ← −Infinity	v ← −Infinity	v ← −Infinity	v ← −Infinity	v ← −Infinity	v ← dQNaN vxisi_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
−NZF	v ← −Infinity	v ← A(p,src2)	v ← p	v ← p	v ← A(p,src2)	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
−Zero	v ← −Infinity	v ← src2	v ← −Zero	v ← Rezd	v ← src2	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Zero	v ← −Infinity	v ← src2	v ← Rezd	v ← +Zero	v ← src2	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+NZF	v ← −Infinity	v ← A(p,src2)	v ← p	v ← p	v ← A(p,src2)	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Infinity	v ← dQNaN vxisi_flag ← 1	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
QNaN & src1 is a NaN	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p vxsnan_flag ← 1
QNaN & src1 not a NaN	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← src2	v ← Q(src2) vxsnan_flag ← 1

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]. For ***xsmaddmdp***, the double-precision floating-point value in doubleword element 0 of VSR[XB].
src3	For ***xsmaddadp***, the double-precision floating-point value in doubleword element 0 of VSR[XB]. For ***xsmaddmdp***, 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. 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 7.25: Actions for xsmadd(a|m)dp

VSX Scalar Multiply-Add Type-A Single-Precision XX3-form

xsmaddasp XT,XA,XB

60	T	A	B	1	AX	BX	TX
0	6	11	16	21	29	30	31

VSX Scalar Multiply-Add Type-M Single-Precision XX3-form

xsmaddmsp XT,XA,XB

60	T	A	B	9	AX	BX	TX
0	6	11	16	21	29	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

reset_xflags()

if ''xsmaddasp'' then do
   src1 ← bfp_CONVERT_FROM_BFP64(VSR[32×AX+A].dword[0])
   src2 ← bfp_CONVERT_FROM_BFP64(VSR[32×TX+T].dword[0])
   src3 ← bfp_CONVERT_FROM_BFP64(VSR[32×BX+B].dword[0])
end
if ''xsmaddmsp'' then do
   src1 ← bfp_CONVERT_FROM_BFP64(VSR[32×AX+A].dword[0])
   src2 ← bfp_CONVERT_FROM_BFP64(VSR[32×BX+B].dword[0])
   src3 ← bfp_CONVERT_FROM_BFP64(VSR[32×TX+T].dword[0])
end

v        ← bfp_MULTIPLY_ADD(src1, src3, src2)
rnd      ← bfp_ROUND_TO_BFP32(FPSCR.RN, v)
result32 ← bfp32_CONVERT_FROM_BFP(rnd)
result64 ← bfp64_CONVERT_FROM_BFP(rnd)

if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
if vximz_flag=1 then SetFX(FPSCR.VXIMZ)
if vxisi_flag=1 then SetFX(FPSCR.VXISI)
if ox_flag=1 then SetFX(FPSCR.OX)
if ux_flag=1 then SetFX(FPSCR.UX)
if xx_flag=1 then SetFX(FPSCR.XX)

vx_flag  ← vxsnan_flag | vximz_flag | vxisi_flag
vex_flag ← FPSCR.VE & vx_flag

if vex_flag=0 then do
   VSR[32×TX+T].dword[0] ← result64
   VSR[32×TX+T].dword[1] ← 0x0000_0000_0000_0000
   FPSCR.FPRF ← fprf_CLASS_BFP32(result32)
   FPSCR.FR   ← inc_flag
   FPSCR.FI   ← xx_flag
end
else do
   FPSCR.FR   ← 0b0
   FPSCR.FI   ← 0b0
end
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

For ***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 ***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[40] by src3, producing a product having unbounded range and precision.

See part 1 of Table 7.26, “Actions for xsmadd(a|m)sp,” on page 653.

src2 is added[41] to the product, producing a sum having unbounded range and precision.

The sum is normalized[42].

See part 2 of Table 7.26, “Actions for xsmadd(a|m)sp,” on page 653.

The intermediate result is rounded to single-precision using the rounding mode specified by RN.

See Table 7.9, “Scalar Floating-Point Intermediate Result Handling,” on page 617.

The result is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are set to 0.

FPRF is set to the class and sign of the result as represented in single-precision format. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.

If a trap-enabled invalid operation exception occurs, VSR[XT] and FPRF are not modified, and FR and FI are set to 0.

[40] Floating-point multiplication is based on exponent addition and multiplication of the significands.

[41] 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.

[42] Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.

See Table 7.10, “VSX Scalar Floating-Point Final Result,” on page 618.

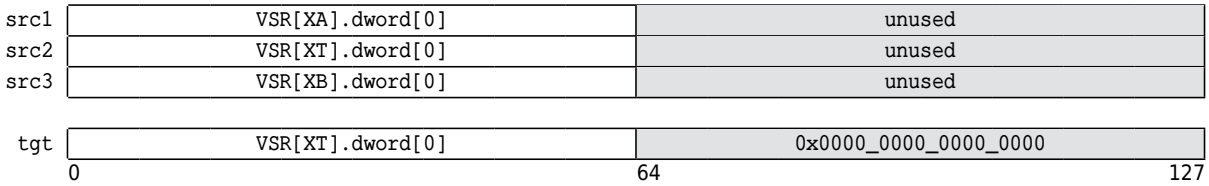
Special Registers Altered:

Register	*Field(s)*
FPSCR	FPRF FR FI FX OX UX XX VXSNAN VXISI VXIMZ

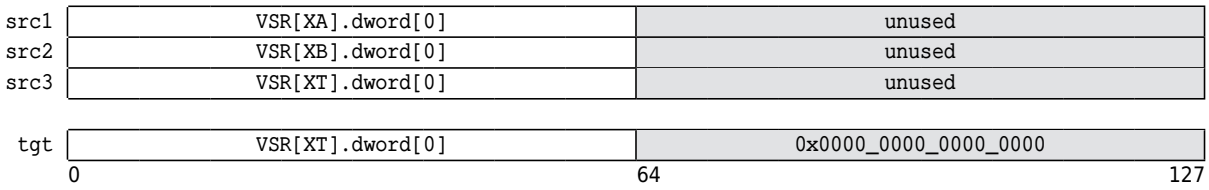
Programming Note

Previous versions of the architecture allowed the contents of doubleword 1 of the result register to be undefined. However, all processors that support this instruction write 0s into doubleword 1 of the result register, as is required by this version of the architecture.

VSR Data Layout for xsmaddasp



VSR Data Layout for xsmaddmsp



Part 1: Multiply		src3							
		−Infinity	−NZF	−Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
src1	−Infinity	p ← +Infinity	p ← +Infinity	p ← dQNaN vximz_flag ← 1	p ← dQNaN vximz_flag ← 1	p ← −Infinity	p ← −Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	−NZF	p ← +Infinity	p ← M(src1,src3)	p ← +Zero	p ← −Zero	p ← M(src1,src3)	p ← −Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	−Zero	p ← dQNaN vximz_flag ← 1	p ← +Zero	p ← +Zero	p ← −Zero	p ← −Zero	p ← dQNaN vximz_flag ← 1	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	+Zero	p ← dQNaN vximz_flag ← 1	p ← −Zero	p ← −Zero	p ← +Zero	p ← +Zero	p ← dQNaN vximz_flag ← 1	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	+NZF	p ← −Infinity	p ← M(src1,src3)	p ← −Zero	p ← +Zero	p ← M(src1,src3)	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	+Infinity	p ← −Infinity	p ← −Infinity	p ← dQNaN vximz_flag ← 1	p ← dQNaN vximz_flag ← 1	p ← +Infinity	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	QNaN	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1 vxsnan_flag ← 1
	SNaN	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1

Part 2: Add		src2							
		−Infinity	−NZF	−Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
p	−Infinity	v ← −Infinity	v ← −Infinity	v ← −Infinity	v ← −Infinity	v ← −Infinity	v ← dQNaN vxisi_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	−NZF	v ← −Infinity	v ← A(p,src2)	v ← p	v ← p	v ← A(p,src2)	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	−Zero	v ← −Infinity	v ← src2	v ← −Zero	v ← Rezd	v ← src2	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+Zero	v ← −Infinity	v ← src2	v ← Rezd	v ← +Zero	v ← src2	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+NZF	v ← −Infinity	v ← A(p,src2)	v ← p	v ← p	v ← A(p,src2)	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+Infinity	v ← dQNaN vxisi_flag ← 1	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	QNaN & src1 is a NaN	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p vxsnan_flag ← 1
	QNaN & src1 not a NaN	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← src2	v ← Q(src2) vxsnan_flag ← 1

Explanation:

src1	The double-precision floating-point value in doubleword element 0 of VSR[XA].
src2	For ***xsmaddasp***, the double-precision floating-point value in doubleword element 0 of VSR[XT]. For ***xsmaddmsp***, the double-precision floating-point value in doubleword element 0 of VSR[XB].
src3	For ***xsmaddasp***, the double-precision floating-point value in doubleword element 0 of VSR[XB]. For ***xsmaddmsp***, 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. 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 7.26: Actions for xsmadd(a|m)sp

VSX Scalar Multiply-Add Quad-Precision [using round to Odd] X-form

xsmaddqp	VRT,VRA,VRB	(RO=0)
xsmaddqpo	VRT,VRA,VRB	(RO=1)

63	VRT	VRA	VRB	388	RO
0	6	11	16	21	31

```
if MSR.VSX=0 then VSX_Unavailable()

reset_xflags()

src1   ← bfp_CONVERT_FROM_BFP128(VSR[VRA+32])
src2   ← bfp_CONVERT_FROM_BFP128(VSR[VRT+32])
src3   ← bfp_CONVERT_FROM_BFP128(VSR[VRB+32])
v      ← bfp_MULTIPLY_ADD(src1, src3, src2)
rnd    ← bfp_ROUND_TO_BFP128(RO, FPSCR.RN, v)
result ← bfp128_CONVERT_FROM_BFP(rnd)

if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
if vximz_flag=1  then SetFX(FPSCR.VXIMZ)
if vxisi_flag=1  then SetFX(FPSCR.VXISI)
if ox_flag=1     then SetFX(FPSCR.OX)
if ux_flag=1     then SetFX(FPSCR.UX)
if xx_flag=1     then SetFX(FPSCR.XX)

vx_flag  ← vxsnan_flag | vximz_flag | vxisi_flag
ex_flag  ← FPSCR.VE & 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 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[43].

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 7.25. "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 7.25. "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 7.9, "Scalar Floating-Point Intermediate Result Handling," on page 617.

The result is placed into VSR[VRT+32] in quad-precision format.

FPRF is set to the class and sign of the result. FR is set to indicate if the rounded result was incremented. FI is set to indicate the result is inexact.

If a trap-disabled Invalid Operation exception occurs, FR and FI are set to 0.

If a trap-enabled Invalid Operation exception occurs, VSR[VRT+32] and FPRF are not modified, and FR and FI are set to 0.

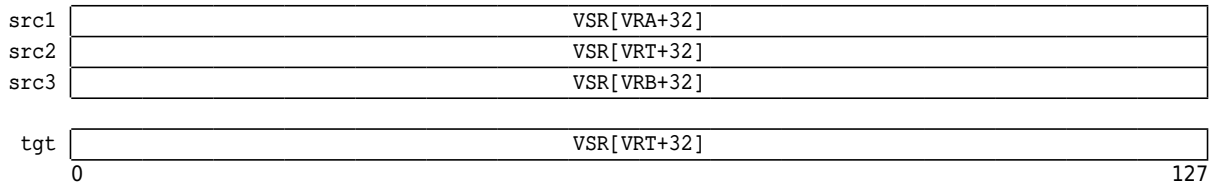
See Table 7.10, "VSX Scalar Floating-Point Final Result," on page 618.

Special Registers Altered:

Register	*Field(s)*
FPSCR	FPRF FR FI FX VXSNAN VXIMZ VXISI OX UX XX

[43] The quad-precision default Quiet NaN is the value, 0x7FFF_8000_0000_0000_0000_0000_0000.

VSR Data Layout for xsmaddqp[o]



Part 1: Multiply	src3: −Infinity	−NZF	−Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
src1: −Infinity	p ← +Infinity		p ← dQNaN vximz_flag ← 1			p ← −Infinity	p ← src3	p ← quiet(src3) vxsnan_flag ← 1
src1: −NZF		p ← mul(src1,src3)			p ← mul(src1,src3)			
src1: −Zero	p ← dQNaN vximz_flag ← 1		p ← +Zero	p ← −Zero		p ← dQNaN vximz_flag ← 1		
src1: +Zero			p ← −Zero	p ← +Zero				
src1: +NZF		p ← mul(src1,src3)			p ← mul(src1,src3)			
src1: +Infinity	p ← −Infinity		p ← dQNaN vximz_flag ← 1			p ← +Infinity		
src1: QNaN	p ← src1							p ← src1 vxsnan_flag ← 1
src1: SNaN	p ← quiet(src1) vxsnan_flag ← 1							

Part 2: Add	src2: −Infinity	−NZF	−Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
p: −Infinity	v ← −Infinity					v ← dQNaN vxisi_flag ← 1	v ← src2	v ← quiet(src2) vxsnan_flag ← 1
p: −NZF		v ← add(p,src2)	v ← p		v ← add(p,src2)			
p: −Zero		v ← src2	v ← −Zero	v ← Rezd	v ← src2			
p: +Zero			v ← Rezd	v ← +Zero				
p: +NZF		v ← add(p,src2)	v ← p		v ← add(p,src2)			
p: +Infinity	v ← dQNaN vxisi_flag ← 1					v ← +Infinity		
p: QNaN & src1 is a NaN	v ← p							v ← p vxsnan_flag ← 1
p: QNaN & src1 not a NaN							v ← src2	v ← quiet(src2) vxsnan_flag ← 1

Explanation:

src1	The quad-precision floating-point value in `VSR[VRA+32]`.
src2	The quad-precision floating-point value in `VSR[VRT+32]`.
src3	The quad-precision floating-point value in `VSR[VRB+32]`.
dQNaN	Default quiet NaN (`0x7FFF_8000_0000_0000_0000_0000_0000`).
NZF	Nonzero finite number.
Rezd	Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs). Can also occur with two nonzero finite number source operands.
quiet(x)	Return a QNaN with the payload of x.
add(x,y)	Return the normalized sum of floating-point value x and floating-point value y, having unbounded range and precision. Note: If x = -y, v is considered to be an exact-zero-difference result (Rezd).
mul(x,y)	Return the normalized product of floating-point value x and floating-point value y, having unbounded range and precision.
p	The intermediate product having unbounded range and precision.
v	The intermediate result having unbounded range and precision.

Table 7.27: Actions for xsmaddqp[o]

VSX Scalar Multiply-Subtract Type-A Double-Precision XX3-form

xsmsubadp XT,XA,XB

60	T	A	B	49	AX	BX	TX
0	6	11	16	21	29	30	31

VSX Scalar Multiply-Subtract Type-M Double-Precision XX3-form

xsmsubmdp XT,XA,XB

60	T	A	B	57	AX	BX	TX
0	6	11	16	21	29	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

reset_xflags()

if ''xsmsubadp'' then do
   src1 ← bfp_CONVERT_FROM_BFP64(VSR[32×AX+A].dword[0])
   src2 ← bfp_CONVERT_FROM_BFP64(VSR[32×TX+T].dword[0])
   src3 ← bfp_CONVERT_FROM_BFP64(VSR[32×BX+B].dword[0])
end
if ''xsmsubmdp'' then do
   src1 ← bfp_CONVERT_FROM_BFP64(VSR[32×AX+A].dword[0])
   src2 ← bfp_CONVERT_FROM_BFP64(VSR[32×BX+B].dword[0])
   src3 ← bfp_CONVERT_FROM_BFP64(VSR[32×TX+T].dword[0])
end

v      ← bfp_MULTIPLY_ADD(src1, src3, bfp_NEGATE(src2))
rnd    ← bfp_ROUND_TO_BFP64(0b0,FPSCR.RN, v)
result ← bfp64_CONVERT_FROM_BFP(rnd)

if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
if vximz_flag=1 then SetFX(FPSCR.VXIMZ)
if vxisi_flag=1 then SetFX(FPSCR.VXISI)
if ox_flag=1 then SetFX(FPSCR.OX)
if ux_flag=1 then SetFX(FPSCR.UX)
if xx_flag=1 then SetFX(FPSCR.XX)

vx_flag  ← vxsnan_flag | vximz_flag | vxisi_flag
vex_flag ← FPSCR.VE & vx_flag

if vex_flag=0 then do
   VSR[32×TX+T].dword[0] ← result
   VSR[32×TX+T].dword[1] ← 0x0000_0000_0000_0000
   FPSCR.FPRF ← fprf_CLASS_BFP64(result)
   FPSCR.FR   ← inc_flag
   FPSCR.FI   ← xx_flag
end
else do
   FPSCR.FR   ← 0b0
   FPSCR.FI   ← 0b0
end
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

For ***xsmsubadp***, 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 ***xsmsubmdp***, 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[44] by src3, producing a product having unbounded range and precision.

See part 1 of Table 7.28.

src2 is negated and added[45] to the product, producing a sum having unbounded range and precision.

The result, having unbounded range and precision, is normalized[46].

See part 2 of Table 7.28.

The intermediate result is rounded to double-precision using the rounding mode specified by RN.

See Table 7.9, “Scalar Floating-Point Intermediate Result Handling,” on page 617.

The result is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are set to 0.

FPRF is set to the class and sign of the result. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.

If a trap-enabled invalid operation exception occurs, VSR[XT] and FPRF are not modified, and FR and FI are set to 0.

See Table 7.10, “VSX Scalar Floating-Point Final Result,” on page 618.

[44] Floating-point multiplication is based on exponent addition and multiplication of the significands.

[45] 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.

[46] 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.

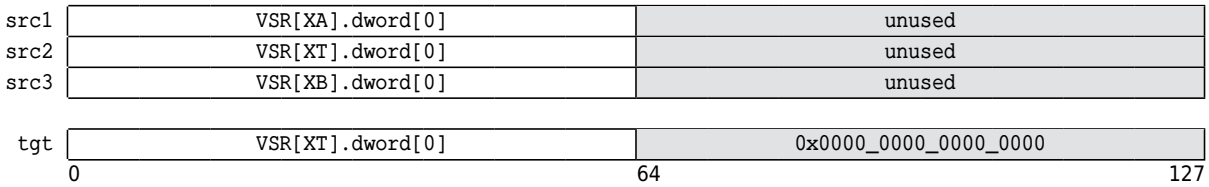
Special Registers Altered:

Register	*Field(s)*
FPSCR	FPRF FR FI FX OX UX XX VXSNAN VXISI VXIMZ

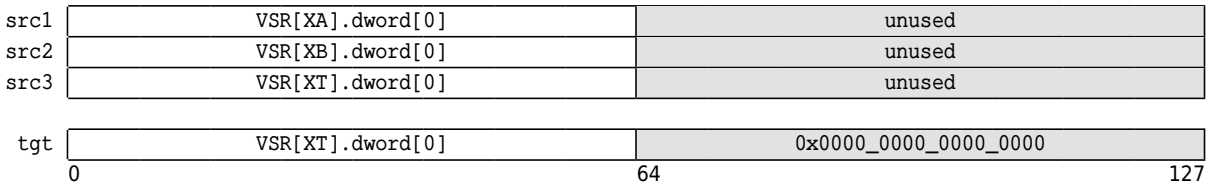
Programming Note

Previous versions of the architecture allowed the contents of doubleword 1 of the result register to be undefined. However, all processors that support this instruction write 0s into doubleword 1 of the result register, as is required by this version of the architecture.

VSR Data Layout for xsmsubadp



VSR Data Layout for xsmsubmdp



Part 1: Multiply		src3							
		−Infinity	−NZF	−Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
src1	−Infinity	p ← +Infinity	p ← +Infinity	p ← dQNaN vximz_flag ← 1	p ← dQNaN vximz_flag ← 1	p ← −Infinity	p ← −Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	−NZF	p ← +Infinity	p ← M(src1,src3)	p ← +Zero	p ← −Zero	p ← M(src1,src3)	p ← −Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	−Zero	p ← dQNaN vximz_flag ← 1	p ← +Zero	p ← +Zero	p ← −Zero	p ← −Zero	p ← dQNaN vximz_flag ← 1	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	+Zero	p ← dQNaN vximz_flag ← 1	p ← −Zero	p ← −Zero	p ← +Zero	p ← +Zero	p ← dQNaN vximz_flag ← 1	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	+NZF	p ← −Infinity	p ← M(src1,src3)	p ← −Zero	p ← +Zero	p ← M(src1,src3)	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	+Infinity	p ← −Infinity	p ← −Infinity	p ← dQNaN vximz_flag ← 1	p ← dQNaN vximz_flag ← 1	p ← +Infinity	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	QNaN	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1 vxsnan_flag ← 1
	SNaN	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1

Part 2: Subtract		src2							
		−Infinity	−NZF	−Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
p	−Infinity	v ← dQNaN vxisi_flag ← 1	v ← −Infinity	v ← −Infinity	v ← −Infinity	v ← −Infinity	v ← −Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	−NZF	v ← +Infinity	v ← S(p,src2)	v ← p	v ← p	v ← S(p,src2)	v ← −Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	−Zero	v ← +Infinity	v ← −src2	v ← Rezd	v ← −Zero	v ← −src2	v ← −Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+Zero	v ← +Infinity	v ← −src2	v ← +Zero	v ← Rezd	v ← −src2	v ← −Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+NZF	v ← +Infinity	v ← S(p,src2)	v ← p	v ← p	v ← S(p,src2)	v ← −Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← dQNaN vxisi_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	QNaN & src1 is a NaN	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p vxsnan_flag ← 1
	QNaN & src1 not a NaN	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← src2	v ← Q(src2) vxsnan_flag ← 1

Explanation:

src1	The double-precision floating-point value in doubleword element 0 of VSR[XA].
src2	For ***xsmsubadp***, the double-precision floating-point value in doubleword element 0 of VSR[XT]. For ***xsmsubmdp***, the double-precision floating-point value in doubleword element 0 of VSR[XB].
src3	For ***xsmsubadp***, the double-precision floating-point value in doubleword element 0 of VSR[XB]. For ***xsmsubmdp***, the double-precision floating-point value in doubleword element 0 of VSR[XT].
dQNaN	Default quiet NaN (`0x7FF8_0000_0000_0000`).
NZF	Nonzero finite number.
Rezd	Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs). Can also occur with two nonzero finite number source operands.
Q(x)	Return a QNaN with the payload of x.
S(x,y)	Return the normalized sum of floating-point value x and negated floating-point value y, having unbounded range and precision. Note: If x = y, v is considered to be an exact-zero-difference result (Rezd).
M(x,y)	Return the normalized product of floating-point value x and floating-point value y, having unbounded range and precision.
p	The intermediate product having unbounded range and precision.
v	The intermediate result having unbounded range and precision.

Table 7.28: Actions for xsmsub(a|m)dp

VSX Scalar Multiply-Subtract Type-A Single-Precision XX3-form

xsmsubasp XT,XA,XB

60	T	A	B	17	AX	BX	TX
0	6	11	16	21	29	30	31

VSX Scalar Multiply-Subtract Type-M Single-Precision XX3-form

xsmsubmsp XT,XA,XB

60	T	A	B	25	AX	BX	TX
0	6	11	16	21	29	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

reset_xflags()

if ''xsmsubasp'' then do
   src1 ← bfp_CONVERT_FROM_BFP64(VSR[32×AX+A].dword[0])
   src2 ← bfp_CONVERT_FROM_BFP64(VSR[32×TX+T].dword[0])
   src3 ← bfp_CONVERT_FROM_BFP64(VSR[32×BX+B].dword[0])
end
if ''xsmsubmsp'' then do
   src1 ← bfp_CONVERT_FROM_BFP64(VSR[32×AX+A].dword[0])
   src2 ← bfp_CONVERT_FROM_BFP64(VSR[32×BX+B].dword[0])
   src3 ← bfp_CONVERT_FROM_BFP64(VSR[32×TX+T].dword[0])
end

v   ← bfp_MULTIPLY_ADD(src1, src3, bfp_NEGATE(src2))
rnd ← bfp_ROUND_TO_BFP32(FPSCR.RN, v)

result32 ← bfp32_CONVERT_FROM_BFP(rnd)
result64 ← bfp64_CONVERT_FROM_BFP(rnd)

if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
if vximz_flag=1 then SetFX(FPSCR.VXIMZ)
if vxisi_flag=1 then SetFX(FPSCR.VXISI)
if ox_flag=1 then SetFX(FPSCR.OX)
if ux_flag=1 then SetFX(FPSCR.UX)
if xx_flag=1 then SetFX(FPSCR.XX)

vx_flag  ← vxsnan_flag | vximz_flag | vxisi_flag
vex_flag ← FPSCR.VE & vx_flag

if vex_flag=0 then do
   VSR[32×TX+T].dword[0] ← result64
   VSR[32×TX+T].dword[1] ← 0x0000_0000_0000_0000
   FPSCR.FPRF ← fprf_CLASS_BFP32(result32)
   FPSCR.FR   ← inc_flag
   FPSCR.FI   ← xx_flag
end
else do
   FPSCR.FR   ← 0b0
   FPSCR.FI   ← 0b0
end
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

For ***xsmsubasp***, 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 ***xsmsubmsp***, 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[47] by src3, producing a product having unbounded range and precision.

See part 1 of Table 7.29, “Actions for xsmsub(a|m)sp”.

src2 is negated and added[48] to the product, producing a sum having unbounded range and precision.

The result, having unbounded range and precision, is normalized[49].

See part 2 of Table 7.29, “Actions for xsmsub(a|m)sp”.

The intermediate result is rounded to single-precision using the rounding mode specified by RN.

See Table 7.9, “Scalar Floating-Point Intermediate Result Handling,” on page 617.

The result is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are set to 0.

FPRF is set to the class and sign of the result as represented in single-precision format. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.

If a trap-enabled invalid operation exception occurs, VSR[XT] and FPRF are not modified, and FR and FI are set to 0.

[47] Floating-point multiplication is based on exponent addition and multiplication of the significands.

[48] 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.

[49] Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.

See Table 7.10, “VSX Scalar Floating-Point Final Result,” on page 618.

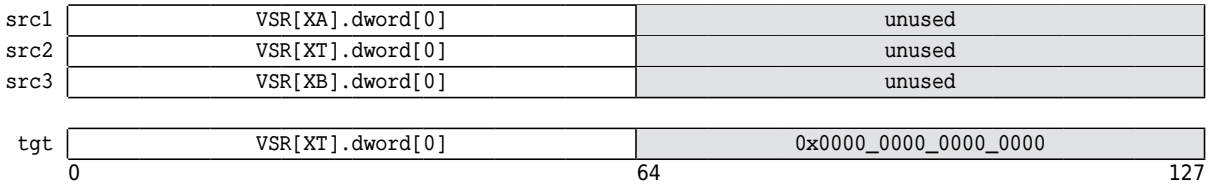
Special Registers Altered:

Register	*Field(s)*
FPSCR	FPRF FR FI FX OX UX XX VXSNAN VXISI VXIMZ

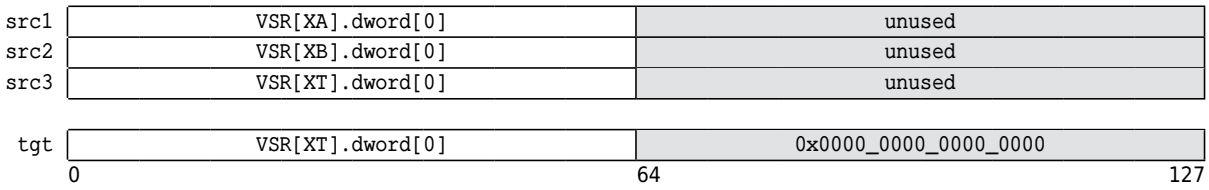
Programming Note

Previous versions of the architecture allowed the contents of doubleword 1 of the result register to be undefined. However, all processors that support this instruction write 0s into doubleword 1 of the result register, as is required by this version of the architecture.

VSR Data Layout for xsmsubasp



VSR Data Layout for xsmsubmsp



Part 1: Multiply	src3							
src1	−Infinity	−NZF	−Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
−Infinity	p ← +Infinity	p ← +Infinity	p ← dQNaN vximz_flag ← 1	p ← dQNaN vximz_flag ← 1	p ← −Infinity	p ← −Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
−NZF	p ← +Infinity	p ← M(src1,src3)	p ← +Zero	p ← −Zero	p ← M(src1,src3)	p ← −Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
−Zero	p ← dQNaN vximz_flag ← 1	p ← +Zero	p ← +Zero	p ← −Zero	p ← −Zero	p ← dQNaN vximz_flag ← 1	p ← src3	p ← Q(src3) vxsnan_flag ← 1
+Zero	p ← dQNaN vximz_flag ← 1	p ← −Zero	p ← −Zero	p ← +Zero	p ← +Zero	p ← dQNaN vximz_flag ← 1	p ← src3	p ← Q(src3) vxsnan_flag ← 1
+NZF	p ← −Infinity	p ← M(src1,src3)	p ← −Zero	p ← +Zero	p ← M(src1,src3)	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
+Infinity	p ← −Infinity	p ← −Infinity	p ← dQNaN vximz_flag ← 1	p ← dQNaN vximz_flag ← 1	p ← +Infinity	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
QNaN	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1 vxsnan_flag ← 1
SNaN	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1

Part 2: Subtract	src2							
p	−Infinity	−NZF	−Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
−Infinity	v ← dQNaN vxisi_flag ← 1	v ← −Infinity	v ← −Infinity	v ← −Infinity	v ← −Infinity	v ← −Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
−NZF	v ← +Infinity	v ← S(p,src2)	v ← p	v ← p	v ← S(p,src2)	v ← −Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
−Zero	v ← +Infinity	v ← −src2	v ← Rezd	v ← −Zero	v ← −src2	v ← −Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Zero	v ← +Infinity	v ← −src2	v ← +Zero	v ← Rezd	v ← −src2	v ← −Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+NZF	v ← +Infinity	v ← S(p,src2)	v ← p	v ← p	v ← S(p,src2)	v ← −Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← dQNaN vxisi_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
QNaN & src1 is a NaN	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p vxsnan_flag ← 1
QNaN & src1 not a NaN	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← src2	v ← Q(src2) vxsnan_flag ← 1

Explanation:

src1	The double-precision floating-point value in doubleword element 0 of VSR[XA].
src2	For ***xsmsubasp***, the double-precision floating-point value in doubleword element 0 of VSR[XT]. For ***xsmsubmsp***, the double-precision floating-point value in doubleword element 0 of VSR[XB].
src3	For ***xsmsubasp***, the double-precision floating-point value in doubleword element 0 of VSR[XB]. For ***xsmsubmsp***, the double-precision floating-point value in doubleword element 0 of VSR[XT].
dQNaN	Default quiet NaN (`0x7FF8_0000_0000_0000`).
NZF	Nonzero finite number.
Rezd	Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs). Can also occur with two nonzero finite number source operands.
Q(x)	Return a QNaN with the payload of x.
S(x,y)	Return the normalized sum of floating-point value x and negated floating-point value y, having unbounded range and precision. Note: If x = y, v is considered to be an exact-zero-difference result (Rezd).
M(x,y)	Return the normalized product of floating-point value x and floating-point value y, having unbounded range and precision.
p	The intermediate product having unbounded range and precision.
v	The intermediate result having unbounded range and precision.

Table 7.29: Actions for xsmsub(a|m)sp

VSX Scalar Multiply-Subtract Quad-Precision [using round to Odd] X-form

xsmsubqp	VRT,VRA,VRB	(RO=0)
xsmsubqpo	VRT,VRA,VRB	(RO=1)

63	VRT	VRA	VRB	420	RO
0	6	11	16	21	31

```
if MSR.VSX=0 then VSX_Unavailable()

reset_xflags()

src1   ← bfp_CONVERT_FROM_BFP128(VSR[VRA+32])
src2   ← bfp_CONVERT_FROM_BFP128(VSR[VRT+32])
src3   ← bfp_CONVERT_FROM_BFP128(VSR[VRB+32])
v      ← bfp_MULTIPLY_ADD(src1, src3, bfp_NEGATE(src2))
rnd    ← bfp_ROUND_TO_BFP128(RO, FPSCR.RN, v)
result ← bfp128_CONVERT_FROM_BFP(rnd)

if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
if vximz_flag=1 then SetFX(FPSCR.VXIMZ)
if vxisi_flag=1 then SetFX(FPSCR.VXISI)
if ox_flag=1 then SetFX(FPSCR.OX)
if ux_flag=1 then SetFX(FPSCR.UX)
if xx_flag=1 then SetFX(FPSCR.XX)

vx_flag  ← vxsnan_flag | vximz_flag | vxisi_flag
vex_flag ← FPSCR.VE & vx_flag

if vex_flag=0 then do
   VSR[VRT+32] ← result
   FPSCR.FPRF  ← fprf_CLASS_BFP128(result)
end
FPSCR.FR ← (vx_flag=0) & inc_flag
FPSCR.FI ← (vx_flag=0) & xx_flag
```

Let src1 be the floating-point value in VSR[VRA+32] represented in quad-precision format.

Let src2 be the floating-point value in VSR[VRT+32] represented in quad-precision format.

Let src3 be the floating-point value in VSR[VRB+32] represented in quad-precision format.

If either src1, src2, or src3 is a Signalling NaN, an Invalid Operation exception occurs and VXSNAN is set to 1.

If src1 is an Infinity value and src3 is a Zero value, or if src1 is a Zero value and src3 is an Infinity value, an Invalid Operation exception occurs and VXIMZ is set to 1.

If src2 and the product of src1 and src3 are Infinity values having same signs, an Invalid Operation exception occurs and VXISI is set to 1.

If src1 is a Signalling NaN, the result is the Quiet NaN corresponding to src1.

Otherwise, if src1 is a Quiet NaN, the result is src1.

Otherwise, if src2 is a Signalling NaN, the result is the Quiet NaN corresponding to src2.

Otherwise, if 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[50].

Otherwise, if the product of src1 and src3, and src2 are Infinity values having same signs, the result is the default Quiet NaN.

Otherwise, do the following.

> src1 is multiplied by src3, producing a product having unbounded significand precision and exponent range.
>
> See part 1 of Table 7.30. “Actions for xsmsubqp[o]”.
>
> src2 is negated and added to the product, producing a sum having unbounded range and precision.
>
> See part 2 of Table 7.30. “Actions for xsmsubqp[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 7.9, “Scalar Floating-Point Intermediate Result Handling,” on page 617.

The result is placed into VSR[VRT+32] in quad-precision format.

FPRF is set to the class and sign of the result. FR is set to indicate if the rounded result was incremented. FI is set to indicate the result is inexact.

If a trap-disabled Invalid Operation exception occurs, FR and FI are set to 0.

If a trap-enabled Invalid Operation exception occurs, VSR[VRT+32] and FPRF are not modified, and FR and FI are set to 0.

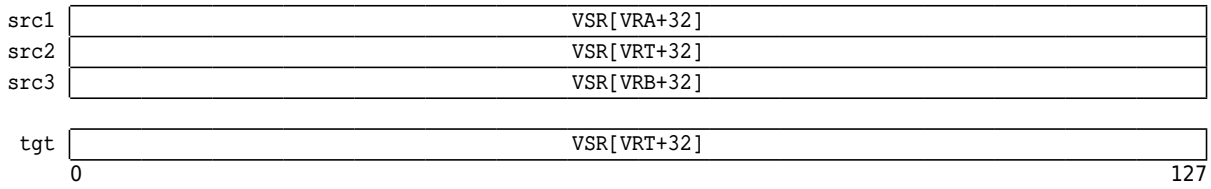
See Table 7.10, “VSX Scalar Floating-Point Final Result,” on page 618.

Special Registers Altered:

Register	*Field(s)*
FPSCR	FPRF FR FI FX VXSNAN VXIMZ VXISI OX UX XX

[50] The quad-precision default Quiet NaN is the value, 0x7FFF_8000_0000_0000_0000_0000_0000.

VSR Data Layout for xsmsubqp[o]



Part 1: Multiply

src1 \ src3	−Infinity	−NZF	−Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
−Infinity	p ← +Infinity		p ← dQNaN vximz_flag ← 1			p ← −Infinity	p ← src3	p ← quiet(src3) vxsnan_flag ← 1
−NZF		p ← mul(src1,src3)			p ← mul(src1,src3)			
−Zero	p ← dQNaN vximz_flag ← 1		p ← +Zero	p ← −Zero		p ← dQNaN vximz_flag ← 1		
+Zero			p ← −Zero	p ← +Zero				
+NZF		p ← mul(src1,src3)			p ← mul(src1,src3)			
+Infinity	p ← −Infinity		p ← dQNaN vximz_flag ← 1			p ← +Infinity		
QNaN	p ← src1							p ← src1 vxsnan_flag ← 1
SNaN	p ← quiet(src1) vxsnan_flag ← 1							

Part 2: Subtract

p \ src2	−Infinity	−NZF	−Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
−Infinity	v ← dQNaN vxisi_flag ← 1					v ← −Infinity	v ← src2	v ← quiet(src2) vxsnan_flag ← 1
−NZF		v ← sub(p,src2)	v ← p		v ← sub(p,src2)			
−Zero		v ← −src2	v ← Rezd	v ← −Zero	v ← −src2			
+Zero			v ← +Zero	v ← Rezd				
+NZF		v ← sub(p,src2)	v ← p		v ← sub(p,src2)			
+Infinity	v ← +Infinity					v ← dQNaN vxisi_flag ← 1		
QNaN & src1 is a NaN	v ← p							v ← p vxsnan_flag ← 1
QNaN & src1 not a NaN							v ← src2	v ← quiet(src2) vxsnan_flag ← 1

Explanation:

src1	The quad-precision floating-point value in `VSR[VRA+32]`.
src2	The quad-precision floating-point value in `VSR[VRT+32]`.
src3	The quad-precision floating-point value in `VSR[VRB+32]`.
dQNaN	Default quiet NaN (`0x7FFF_8000_0000_0000_0000_0000_0000`).
NZF	Nonzero finite number.
Rezd	Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs). Can also occur with two nonzero finite number source operands.
quiet(x)	Return a QNaN with the payload of x.
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 7.30: Actions for xsmsubqp[o]

VSX Scalar Negative Multiply-Add Type-A Double-Precision XX3-form

xsnmaddadp XT,XA,XB

60	T	A	B	161	AX	BX	TX
0	6	11	16	21	29	30	31

VSX Scalar Negative Multiply-Add Type-M Double-Precision XX3-form

xsnmaddmdp XT,XA,XB

60	T	A	B	169	AX	BX	TX
0	6	11	16	21	29	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

reset_xflags()

if ''xsnmaddadp'' then do
   src1 ← bfp_CONVERT_FROM_BFP64(VSR[32×AX+A].dword[0])
   src2 ← bfp_CONVERT_FROM_BFP64(VSR[32×TX+T].dword[0])
   src3 ← bfp_CONVERT_FROM_BFP64(VSR[32×BX+B].dword[0])
end
if ''xsnmaddmdp'' then do
   src1 ← bfp_CONVERT_FROM_BFP64(VSR[32×AX+A].dword[0])
   src2 ← bfp_CONVERT_FROM_BFP64(VSR[32×BX+B].dword[0])
   src3 ← bfp_CONVERT_FROM_BFP64(VSR[32×TX+T].dword[0])
end

v   ← bfp_MULTIPLY_ADD(src1, src3, src2)
rnd ← bfp_NEGATE(bfp_ROUND_TO_BFP64(0b0,FPSCR.RN, v))

result ← bfp64_CONVERT_FROM_BFP(rnd)

if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
if vximz_flag=1 then SetFX(FPSCR.VXIMZ)
if vxisi_flag=1 then SetFX(FPSCR.VXISI)
if ox_flag=1 then SetFX(FPSCR.OX)
if ux_flag=1 then SetFX(FPSCR.UX)
if xx_flag=1 then SetFX(FPSCR.XX)

vx_flag  ← vxsnan_flag | vximz_flag | vxisi_flag
vex_flag ← FPSCR.VE & vx_flag

if vex_flag=0 then do
   VSR[32×TX+T].dword[0] ← result
   VSR[32×TX+T].dword[1] ← 0x0000_0000_0000_0000
   FPSCR.FPRF ← fprf_CLASS_BFP32(result)
   FPSCR.FR   ← inc_flag
   FPSCR.FI   ← xx_flag
end
else do
   FPSCR.FR   ← 0b0
   FPSCR.FI   ← 0b0
end
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

Let src1 be the double-precision floating-point value in doubleword element 0 of VSR[XA].

For ***xsnmaddadp***, do the following.

- Let src2 be the double-precision floating-point value in doubleword element 0 of VSR[XT].
- Let src3 be the double-precision floating-point value in doubleword element 0 of VSR[XB].

For ***xsnmaddmdp***, do the following.

- Let src2 be the double-precision floating-point value in doubleword element 0 of VSR[XB].
- Let src3 be the double-precision floating-point value in doubleword element 0 of VSR[XT].

src1 is multiplied[51] by src3, producing a product having unbounded range and precision.

See part 1 of Table 7.31.

src2 is added[52] to the product, producing a sum having unbounded range and precision.

The sum is normalized[53].

See part 2 of Table 7.31.

The intermediate result is rounded to double-precision using the rounding mode specified by RN.

See Table 7.9, “Scalar Floating-Point Intermediate Result Handling,” on page 617.

The result is negated and placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are set to 0.

FPRF is set to the class and sign of the result. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.

If a trap-enabled invalid operation exception occurs, VSR[XT] and FPRF are not modified, and FR and FI are set to 0.

See Table 7.32, “Scalar Floating-Point Final Result with Negation,” on page 669.

Special Registers Altered:

[51] Floating-point multiplication is based on exponent addition and multiplication of the significands.

[52] 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.

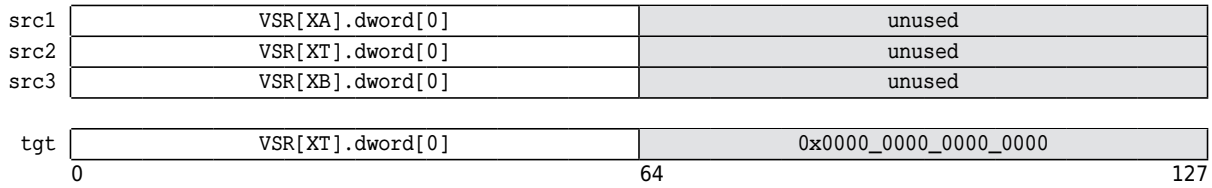
[53] Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.

Register	*Field(s)*
FPSCR	FPRF FR FI FX OX UX XX VXSNAN VXISI VXIMZ

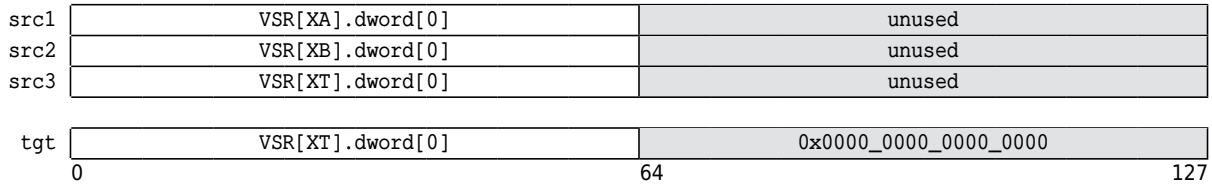
Programming Note

Previous versions of the architecture allowed the contents of doubleword 1 of the result register to be undefined. However, all processors that support this instruction write 0s into doubleword 1 of the result register, as is required by this version of the architecture.

VSR Data Layout for xsnmaddadp



VSR Data Layout for xsnmaddmdp



Part 1: Multiply	src3							
src1	−Infinity	−NZF	−Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
−Infinity	p ← +Infinity	p ← +Infinity	p ← dQNaN vximz_flag ← 1	p ← dQNaN vximz_flag ← 1	p ← −Infinity	p ← −Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
−NZF	p ← +Infinity	p ← M(src1,src3)	p ← +Zero	p ← −Zero	p ← M(src1,src3)	p ← −Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
−Zero	p ← dQNaN vximz_flag ← 1	p ← +Zero	p ← +Zero	p ← −Zero	p ← −Zero	p ← dQNaN vximz_flag ← 1	p ← src3	p ← Q(src3) vxsnan_flag ← 1
+Zero	p ← dQNaN vximz_flag ← 1	p ← −Zero	p ← −Zero	p ← +Zero	p ← +Zero	p ← dQNaN vximz_flag ← 1	p ← src3	p ← Q(src3) vxsnan_flag ← 1
+NZF	p ← −Infinity	p ← M(src1,src3)	p ← −Zero	p ← +Zero	p ← M(src1,src3)	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
+Infinity	p ← −Infinity	p ← −Infinity	p ← dQNaN vximz_flag ← 1	p ← dQNaN vximz_flag ← 1	p ← +Infinity	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
QNaN	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1 vxsnan_flag ← 1
SNaN	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1

Part 2: Add	src2							
p	−Infinity	−NZF	−Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
−Infinity	v ← −Infinity	v ← −Infinity	v ← −Infinity	v ← −Infinity	v ← −Infinity	v ← dQNaN vxisi_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
−NZF	v ← −Infinity	v ← A(p,src2)	v ← p	v ← p	v ← A(p,src2)	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
−Zero	v ← −Infinity	v ← src2	v ← −Zero	v ← Rezd	v ← src2	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Zero	v ← −Infinity	v ← src2	v ← Rezd	v ← +Zero	v ← src2	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+NZF	v ← −Infinity	v ← A(p,src2)	v ← p	v ← p	v ← A(p,src2)	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Infinity	v ← dQNaN vxisi_flag ← 1	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
QNaN & src1 is a NaN	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p vxsnan_flag ← 1
QNaN & src1 not a NaN	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← src2	v ← Q(src2) vxsnan_flag ← 1

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]. For ***xsnmaddmdp***, the double-precision floating-point value in doubleword element 0 of VSR[XB].
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. 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 7.31: Actions for xsnmadd(a|m)dp

Case	FPSCR.VE	FPSCR.OE	FPSCR.UE	FPSCR.ZE	FPSCR.XE	vxsnan_flag	vximz_flag	vxisi_flag	Is r inexact? (r ≠ v)	Is r incremented? (|r| > |v|)	Is q inexact? (q ≠ v)	Is q incremented? (|q| > |v|)	Returned Results and Status Setting
Special	–	–	–	–	–	0	0	0	–	–	–	–	T(n(r)), fprf(class(r)), fi(0), fr(0)
	0	–	–	–	–	–	–	1	–	–	–	–	T(r), fprf(class(r)), fi(0), fr(0), fx(VXISI)
	0	–	–	–	–	0	1	–	–	–	–	–	T(r), fprf(class(r)), fi(0), fr(0), fx(VXIMZ)
	0	–	–	–	–	1	0	–	–	–	–	–	T(r), fprf(class(r)), fi(0), fr(0), fx(VXSNAN)
	0	–	–	–	–	1	1	–	–	–	–	–	T(r), fprf(class(r)), fi(0), fr(0), fx(VXSNAN), fx(VXIMZ)
	1	–	–	–	–	–	–	1	–	–	–	–	fx(VXISI), error()
	1	–	–	–	–	0	1	–	–	–	–	–	fx(VXIMZ), error()
	1	–	–	–	–	1	0	–	–	–	–	–	fx(VXSNAN), error()
	1	–	–	–	–	1	1	–	–	–	–	–	fx(VXSNAN), fx(VXIMZ), error()
Normal	–	–	–	–	–	–	–	–	no	–	–	–	T(n(r)), fprf(class(n(r))), fi(0), fr(0)
	–	–	–	–	0	–	–	–	yes	no	–	–	T(n(r)), fprf(class(n(r))), fi(1), fr(0), fx(XX)
	–	–	–	–	0	–	–	–	yes	yes	–	–	T(n(r)), fprf(class(n(r))), fi(1), fr(1), fx(XX)
	–	–	–	–	1	–	–	–	yes	no	–	–	T(n(r)), fprf(class(n(r))), fi(1), fr(0), fx(XX), error()
	–	–	–	–	1	–	–	–	yes	yes	–	–	T(n(r)), fprf(class(n(r))), fi(1), fr(1), fx(XX), error()
Overflow	–	0	–	–	0	–	–	–	–	–	–	–	T(n(r)), fprf(class(n(r))), fi(1), fr(?), fx(OX), fx(XX)
	–	0	–	–	1	–	–	–	–	–	–	–	T(n(r)), fprf(class(n(r))), fi(1), fr(?), fx(OX), fx(XX), error()
	–	1	–	–	–	–	–	–	–	–	no	–	T(n(q)÷β), fprf(class(n(q)÷β)), fi(0), fr(0), fx(OX), error()
	–	1	–	–	–	–	–	–	–	–	yes	no	T(n(q)÷β), fprf(class(n(q)÷β)), fi(1), fr(0), fx(OX), fx(XX), error()
	–	1	–	–	–	–	–	–	–	–	yes	yes	T(n(q)÷β), fprf(class(n(q)÷β)), fi(1), fr(1), fx(OX), fx(XX), error()
Tiny	–	–	0	–	–	–	–	–	no	–	–	–	T(n(r)), fprf(class(n(r))), fi(0), fr(0)
	–	–	0	–	0	–	–	–	yes	no	–	–	T(n(r)), fprf(class(n(r))), fi(1), fr(0), fx(UX), fx(XX)
	–	–	0	–	0	–	–	–	yes	yes	–	–	T(n(r)), fprf(class(n(r))), fi(1), fr(1), fx(UX), fx(XX)
	–	–	0	–	1	–	–	–	yes	no	–	–	T(n(r)), fprf(class(n(r))), fi(1), fr(0), fx(UX), fx(XX), error()
	–	–	0	–	1	–	–	–	yes	yes	–	–	T(n(r)), fprf(class(n(r))), fi(1), fr(1), fx(UX), fx(XX), error()
	–	–	1	–	–	–	–	–	yes	–	no	–	T(n(q)×β), fprf(class(n(q)×β)), fi(0), fr(0), fx(UX), error()
	–	–	1	–	–	–	–	–	yes	–	yes	no	T(n(q)×β), fprf(class(n(q)×β)), fi(1), fr(0), fx(UX), fx(XX), error()
	–	–	1	–	–	–	–	–	yes	–	yes	yes	T(n(q)×β), fprf(class(n(q)×β)), fi(1), fr(1), fx(UX), fx(XX), error()

Explanation:

–	The results do not depend on this condition.
class(x)	Classifies the floating-point value x as defined in Table 7.1, “Floating-Point Result Flags,” on page 489.
error()	The system error handler is invoked for the trap-enabled exception if MSR.FE0 and MSR.FE1 are set to any mode other than the ignore-exception mode.
fi(x)	FPSCR.FI is set to the value x.
fprf(x)	FPSCR.FPRF is set to the 5-bit value x.
fr(x)	FPSCR.FR is set to the value x.
fx(x)	FPSCR.FX is set to 1 if FPSCR.x=0. FPSCR.x is set to 1.
β	Wrap adjust, where $\beta = 2^{1536}$ for double-precision and $\beta = 2^{192}$ for single-precision.
q	The value defined in Table 7.9, “Scalar Floating-Point Intermediate Result Handling,” on page 617, signficand rounded to the target precision, unbounded exponent range.
r	The value defined in Table 7.9, “Scalar Floating-Point Intermediate Result Handling,” on page 617, signficand rounded to the target precision, bounded exponent range.
v	The precise intermediate result defined in the instruction having unbounded signficand precision, unbounded exponent range.
n(x)	The value x is is negated by complementing the sign bit of x.
T(x)	The value x is placed in element 0 of VSR[XT] in the target precision format. The contents of the remaining element(s) of VSR[XT] are set to 0.

Table 7.32: Scalar Floating-Point Final Result with Negation

VSX Scalar Negative Multiply-Add Type-A Single-Precision XX3-form

xsnmaddasp XT,XA,XB

60	T	A	B	129	AX	BX	TX
0	6	11	16	21	29	30	31

VSX Scalar Negative Multiply-Add Type-M Single-Precision XX3-form

xsnmaddmsp XT,XA,XB

60	T	A	B	137	AX	BX	TX
0	6	11	16	21	29	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

reset_xflags()

if ''xsnmaddasp'' then do
   src1 ← bfp_CONVERT_FROM_BFP64(VSR[32×AX+A].dword[0])
   src2 ← bfp_CONVERT_FROM_BFP64(VSR[32×TX+T].dword[0])
   src3 ← bfp_CONVERT_FROM_BFP64(VSR[32×BX+B].dword[0])
end
if ''xsnmaddmsp'' then do
   src1 ← bfp_CONVERT_FROM_BFP64(VSR[32×AX+A].dword[0])
   src2 ← bfp_CONVERT_FROM_BFP64(VSR[32×BX+B].dword[0])
   src3 ← bfp_CONVERT_FROM_BFP64(VSR[32×TX+T].dword[0])
end

v        ← bfp_MULTIPLY_ADD(src1, src3, src2)
rnd      ← bfp_NEGATE(bfp_ROUND_TO_BFP32(FPSCR.RN, v))
result32 ← bfp32_CONVERT_FROM_BFP(rnd)
result64 ← bfp64_CONVERT_FROM_BFP(rnd)

if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
if vximz_flag=1 then SetFX(FPSCR.VXIMZ)
if vxisi_flag=1 then SetFX(FPSCR.VXISI)
if ox_flag=1 then SetFX(FPSCR.OX)
if ux_flag=1 then SetFX(FPSCR.UX)
if xx_flag=1 then SetFX(FPSCR.XX)

vx_flag  ← vxsnan_flag | vximz_flag | vxisi_flag
vex_flag ← FPSCR.VE & vx_flag

if vex_flag=0 then do
   VSR[32×TX+T].dword[0] ← result64
   VSR[32×TX+T].dword[1] ← 0x0000_0000_0000_0000
   FPSCR.FPRF ← fprf_CLASS_BFP32(result32)
   FPSCR.FR   ← inc_flag
   FPSCR.FI   ← xx_flag
end
else do
   FPSCR.FR   ← 0b0
   FPSCR.FI   ← 0b0
end
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

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[54] by src3, producing a product having unbounded range and precision.

See part 1 of Table 7.33, “Actions for xsnmadd(a|m)sp,” on page 672.

src2 is added[55] to the product, producing a sum having unbounded range and precision.

The sum is normalized[56].

See part 2 of Table 7.33, “Actions for xsnmadd(a|m)sp,” on page 672.

The intermediate result is rounded to single-precision using the rounding mode specified by RN.

See Table 7.9, “Scalar Floating-Point Intermediate Result Handling,” on page 617.

The result is negated and placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are set to 0.

FPRF is set to the class and sign of the result as represented in single-precision format. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.

If a trap-enabled invalid operation exception occurs, VSR[XT] and FPRF are not modified, and FR and FI are set to 0.

[54] Floating-point multiplication is based on exponent addition and multiplication of the significands.

[55] 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.

[56] Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.

See Table 7.32, “Scalar Floating-Point Final Result with Negation,” on page 669.

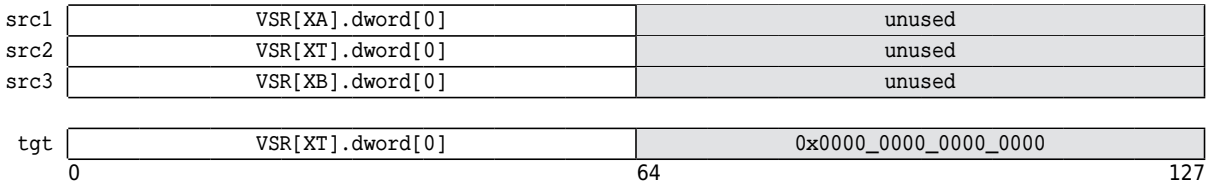
Special Registers Altered:

Register	*Field(s)*
FPSCR	FPRF FR FI FX OX UX XX VXSNAN VXISI VXIMZ

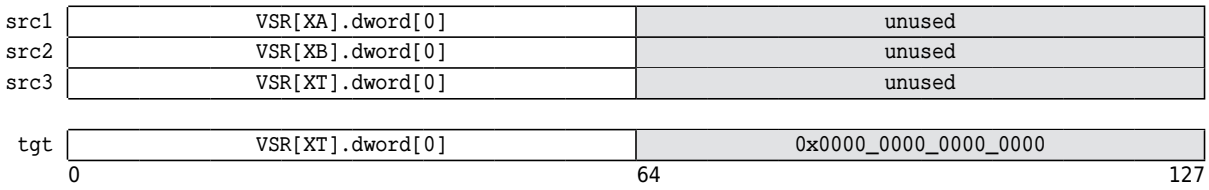
Programming Note

Previous versions of the architecture allowed the contents of doubleword 1 of the result register to be undefined. However, all processors that support this instruction write 0s into doubleword 1 of the result register, as is required by this version of the architecture.

VSR Data Layout for xsnmaddasp



VSR Data Layout for xsnmaddmsp



Part 1: Multiply	src3							
src1	−Infinity	−NZF	−Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
−Infinity	p ← +Infinity	p ← +Infinity	p ← dQNaN vximz_flag ← 1	p ← dQNaN vximz_flag ← 1	p ← −Infinity	p ← −Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
−NZF	p ← +Infinity	p ← M(src1,src3)	p ← +Zero	p ← −Zero	p ← M(src1,src3)	p ← −Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
−Zero	p ← dQNaN vximz_flag ← 1	p ← +Zero	p ← +Zero	p ← −Zero	p ← −Zero	p ← dQNaN vximz_flag ← 1	p ← src3	p ← Q(src3) vxsnan_flag ← 1
+Zero	p ← dQNaN vximz_flag ← 1	p ← −Zero	p ← −Zero	p ← +Zero	p ← +Zero	p ← dQNaN vximz_flag ← 1	p ← src3	p ← Q(src3) vxsnan_flag ← 1
+NZF	p ← −Infinity	p ← M(src1,src3)	p ← −Zero	p ← +Zero	p ← M(src1,src3)	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
+Infinity	p ← −Infinity	p ← −Infinity	p ← dQNaN vximz_flag ← 1	p ← dQNaN vximz_flag ← 1	p ← +Infinity	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
QNaN	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1 vxsnan_flag ← 1
SNaN	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1

Part 2: Add	src2							
p	−Infinity	−NZF	−Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
−Infinity	v ← −Infinity	v ← −Infinity	v ← −Infinity	v ← −Infinity	v ← −Infinity	v ← dQNaN vxisi_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
−NZF	v ← −Infinity	v ← A(p,src2)	v ← p	v ← p	v ← A(p,src2)	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
−Zero	v ← −Infinity	v ← src2	v ← −Zero	v ← Rezd	v ← src2	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Zero	v ← −Infinity	v ← src2	v ← Rezd	v ← +Zero	v ← src2	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+NZF	v ← −Infinity	v ← A(p,src2)	v ← p	v ← p	v ← A(p,src2)	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Infinity	v ← dQNaN vxisi_flag ← 1	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
QNaN & src1 is a NaN	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p vxsnan_flag ← 1
QNaN & src1 not a NaN	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← src2	v ← Q(src2) vxsnan_flag ← 1

Explanation:

src1	The double-precision floating-point value in doubleword element 0 of VSR[XA].
src2	For ***xsnmaddasp***, 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 ***xsnmaddasp***, 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 (`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.

Table 7.33: Actions for xsnmadd(a|m)sp

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)

63	VRT	VRA	VRB	452	RO
0	6	11	16	21	31

```
if MSR.VSX=0 then VSX_Unavailable()

reset_xflags()

src1   ← bfp_CONVERT_FROM_BFP128(VSR[VRA+32])
src2   ← bfp_CONVERT_FROM_BFP128(VSR[VRT+32])
src3   ← bfp_CONVERT_FROM_BFP128(VSR[VRB+32])
v      ← bfp_MULTIPLY_ADD(src1,src3,src2)
rnd    ← bfp_NEGATE(bfp_ROUND_TO_BFP128(RO, FPSCR.RN,
v))
result ← bfp128_CONVERT_FROM_BFP(rnd)

if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
if vximz_flag=1 then SetFX(FPSCR.VXIMZ)
if vxisi_flag=1 then SetFX(FPSCR.VXISI)
if ox_flag=1 then SetFX(FPSCR.OX)
if ux_flag=1 then SetFX(FPSCR.UX)
if xx_flag=1 then SetFX(FPSCR.XX)

vx_flag  ← vxsnan_flag | vximz_flag | vxisi_flag
vex_flag ← FPSCR.VE & vx_flag

if vex_flag=0 then do
   VSR[VRT+32] ← result
   FPSCR.FPRF  ← fprf_CLASS_BFP128(result)
end
FPSCR.FR ← (vx_flag=0) & inc_flag
FPSCR.FI ← (vx_flag=0) & xx_flag
```

Let src1 be the floating-point value in VSR[VRA+32] represented in quad-precision format.

Let src2 be the floating-point value in VSR[VRT+32] represented in quad-precision format.

Let src3 be the floating-point value in VSR[VRB+32] represented in quad-precision format.

If either src1, src2, or src3 is a Signalling NaN, an Invalid Operation exception occurs and VXSNAN is set to 1.

If src1 is an Infinity value and src3 is a Zero value, or if src1 is a Zero value and src3 is an Infinity value, an Invalid Operation exception occurs and VXIMZ is set to 1.

If src2 and the product of src1 and src3 are Infinity values having 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[57].

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 7.25. "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 7.25. "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 7.9, "Scalar Floating-Point Intermediate Result Handling," on page 617.

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.

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.

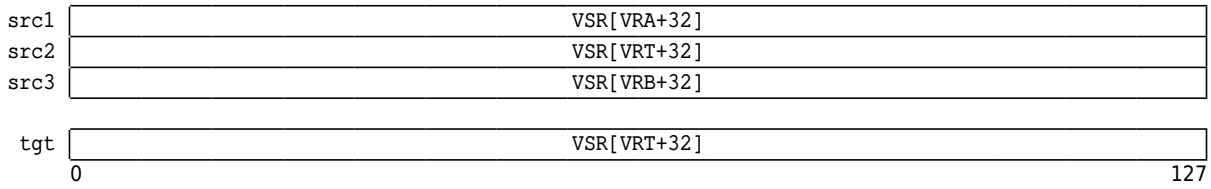
See Table 7.10, "VSX Scalar Floating-Point Final Result," on page 618.

Special Registers Altered:

Register	*Field(s)*
FPSCR	FPRF FR FI FX VXSNAN VXIMZ VXISI OX UX XX

[57] The quad-precision default Quiet NaN is the value, 0x7FFF_8000_0000_0000_0000_0000_0000.

VSR Data Layout for xsnmaddqp[o]



Part 1: Multiply

src1 \ src3	−Infinity	−NZF	−Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
−Infinity	p ← +Infinity		p ← dQNaN vximz_flag ← 1			p ← −Infinity	p ← src3	p ← quiet(src3) vxsnan_flag ← 1
−NZF		p ← mul(src1,src3)			p ← mul(src1,src3)			
−Zero	p ← dQNaN vximz_flag ← 1		p ← +Zero	p ← −Zero		p ← dQNaN vximz_flag ← 1		
+Zero			p ← −Zero	p ← +Zero				
+NZF		p ← mul(src1,src3)			p ← mul(src1,src3)			
+Infinity	p ← −Infinity		p ← dQNaN vximz_flag ← 1			p ← +Infinity		
QNaN	p ← src1							p ← src1 vxsnan_flag ← 1
SNaN	p ← quiet(src1) vxsnan_flag ← 1							

Part 2: Add

p \ src2	−Infinity	−NZF	−Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
−Infinity	v ← −Infinity					v ← dQNaN vxisi_flag ← 1	v ← src2	v ← quiet(src2) vxsnan_flag ← 1
−NZF		v ← add(p,src2)	v ← p		v ← add(p,src2)			
−Zero		v ← src2	v ← −Zero	v ← Rezd	v ← src2			
+Zero			v ← Rezd	v ← +Zero				
+NZF		v ← add(p,src2)	v ← p		v ← add(p,src2)			
+Infinity	v ← dQNaN vxisi_flag ← 1					v ← +Infinity		
QNaN & src1 is a NaN	v ← p							v ← p vxsnan_flag ← 1
QNaN & src1 not a NaN							v ← src2	v ← quiet(src2) vxsnan_flag ← 1

Explanation:

src1	The quad-precision floating-point value in `VSR[VRA+32]`.
src2	The quad-precision floating-point value in `VSR[VRT+32]`.
src3	The quad-precision floating-point value in `VSR[VRB+32]`.
dQNaN	Default quiet NaN (`0x7FFF_8000_0000_0000_0000_0000_0000`).
NZF	Nonzero finite number.
Rezd	Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs). Can also occur with two nonzero finite number source operands.
quiet(x)	Return a QNaN with the payload of x.
add(x,y)	Return the normalized sum of floating-point value x and floating-point value y, having unbounded range and precision. Note: If x = -y, v is considered to be an exact-zero-difference result (Rezd).
mul(x,y)	Return the normalized product of floating-point value x and floating-point value y, having unbounded range and precision.
p	The intermediate product having unbounded range and precision.
v	The intermediate result having unbounded range and precision.

Table 7.34: Actions for xsnmaddqp[o]

VSX Scalar Negative Multiply-Subtract Type-A Double-Precision XX3-form

xsnmsubadp XT,XA,XB

60	T	A	B	177	AX	BX	TX
0	6	11	16	21	29	30	31

VSX Scalar Negative Multiply-Subtract Type-M Double-Precision XX3-form

xsnmsubmdp XT,XA,XB

60	T	A	B	185	AX	BX	TX
0	6	11	16	21	29	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

reset_xflags()

if ''xsnmsubadp'' then do
   src1 ← bfp_CONVERT_FROM_BFP64(VSR[32×AX+A].dword[0])
   src2 ← bfp_CONVERT_FROM_BFP64(VSR[32×TX+T].dword[0])
   src3 ← bfp_CONVERT_FROM_BFP64(VSR[32×BX+B].dword[0])
end
if ''xsnmsubmdp'' then do
   src1 ← bfp_CONVERT_FROM_BFP64(VSR[32×AX+A].dword[0])
   src2 ← bfp_CONVERT_FROM_BFP64(VSR[32×BX+B].dword[0])
   src3 ← bfp_CONVERT_FROM_BFP64(VSR[32×TX+T].dword[0])
end

v      ← bfp_MULTIPLY_ADD(src1, src3, bfp_NEGATE(src2))
rnd    ← bfp_NEGATE(bfp_ROUND_TO_BFP64(0b0,FPSCR.RN,v))
result ← bfp64_CONVERT_FROM_BFP(rnd)

if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
if vximz_flag=1 then SetFX(FPSCR.VXIMZ)
if vxisi_flag=1 then SetFX(FPSCR.VXISI)
if ox_flag=1 then SetFX(FPSCR.OX)
if ux_flag=1 then SetFX(FPSCR.UX)
if xx_flag=1 then SetFX(FPSCR.XX)

vx_flag  ← vxsnan_flag | vximz_flag | vxisi_flag
vex_flag ← FPSCR.VE & vx_flag

if vex_flag=0 then do
   VSR[32×TX+T].dword[0] ← result
   VSR[32×TX+T].dword[1] ← 0x0000_0000_0000_0000
   FPSCR.FPRF ← fprf_CLASS_BFP64(result)
   FPSCR.FR   ← inc_flag
   FPSCR.FI   ← xx_flag
end
else do
   FPSCR.FR   ← 0b0
   FPSCR.FI   ← 0b0
end
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

Let src1 be the double-precision floating-point value in doubleword element 0 of VSR[XA].

For ***xsnmsubadp***, do the following.

- Let src2 be the double-precision floating-point value in doubleword element 0 of VSR[XT].
- Let src3 be the double-precision floating-point value in doubleword element 0 of VSR[XB].

For ***xsnmsubmdp***, 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[58] by src3, producing a product having unbounded range and precision.

See part 1 of Table 7.35.

src2 is negated and added[59] to the product, producing a sum having unbounded range and precision.

The sum is normalized[60].

See part 2 of Table 7.35.

The intermediate result is rounded to double-precision using the rounding mode specified by RN.

See Table 7.9, “Scalar Floating-Point Intermediate Result Handling,” on page 617.

The result is negated and placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are set to 0.

FPRF is set to the class and sign of the result. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.

If a trap-enabled invalid operation exception occurs, VSR[XT] and FPRF are not modified, and FR and FI are set to 0.

See Table 7.32, “Scalar Floating-Point Final Result with Negation,” on page 669.

Special Registers Altered:

[58] Floating-point multiplication is based on exponent addition and multiplication of the significands.

[59] 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.

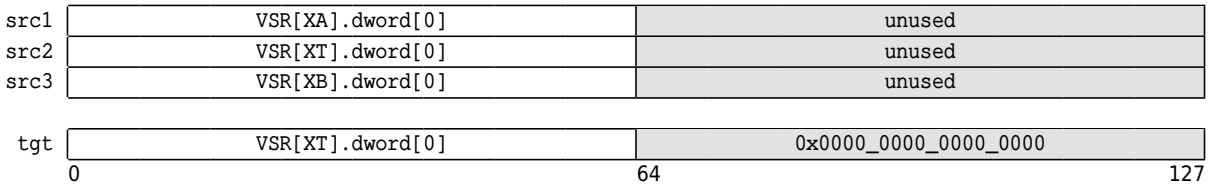
[60] Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.

Register	*Field(s)*
FPSCR	FPRF FR FI FX OX UX XX VXSNAN VXISI VXIMZ

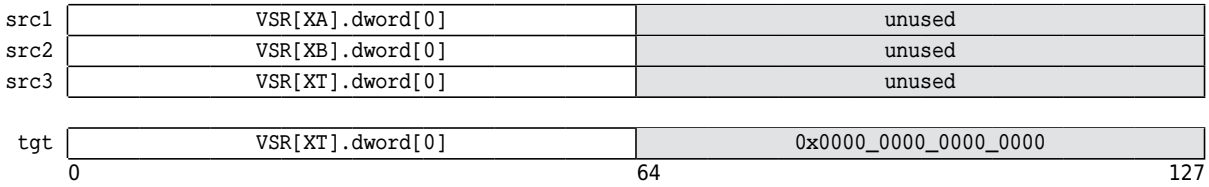
Programming Note

Previous versions of the architecture allowed the contents of doubleword 1 of the result register to be undefined. However, all processors that support this instruction write 0s into doubleword 1 of the result register, as is required by this version of the architecture.

VSR Data Layout for xsnmsubadp



VSR Data Layout for xsnmsubmdp



Part 1: Multiply	src1 \ src3	−Infinity	−NZF	−Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
	−Infinity	p ← +Infinity	p ← +Infinity	p ← dQNaN vximz_flag ← 1	p ← dQNaN vximz_flag ← 1	p ← −Infinity	p ← −Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	−NZF	p ← +Infinity	p ← M(src1,src3)	p ← +Zero	p ← −Zero	p ← M(src1,src3)	p ← −Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	−Zero	p ← dQNaN vximz_flag ← 1	p ← +Zero	p ← +Zero	p ← −Zero	p ← −Zero	p ← dQNaN vximz_flag ← 1	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	+Zero	p ← dQNaN vximz_flag ← 1	p ← −Zero	p ← −Zero	p ← +Zero	p ← +Zero	p ← dQNaN vximz_flag ← 1	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	+NZF	p ← −Infinity	p ← M(src1,src3)	p ← −Zero	p ← +Zero	p ← M(src1,src3)	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	+Infinity	p ← −Infinity	p ← −Infinity	p ← dQNaN vximz_flag ← 1	p ← dQNaN vximz_flag ← 1	p ← +Infinity	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	QNaN	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1 vxsnan_flag ← 1
	SNaN	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1

Part 2: Subtract	p \ src2	−Infinity	−NZF	−Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
	−Infinity	v ← dQNaN vxisi_flag ← 1	v ← −Infinity	v ← −Infinity	v ← −Infinity	v ← −Infinity	v ← −Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	−NZF	v ← +Infinity	v ← S(p,src2)	v ← p	v ← p	v ← S(p,src2)	v ← −Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	−Zero	v ← +Infinity	v ← −src2	v ← Rezd	v ← −Zero	v ← −src2	v ← −Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+Zero	v ← +Infinity	v ← −src2	v ← +Zero	v ← Rezd	v ← −src2	v ← −Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+NZF	v ← +Infinity	v ← S(p,src2)	v ← p	v ← p	v ← S(p,src2)	v ← −Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← dQNaN vxisi_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	QNaN & src1 is a NaN	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p vxsnan_flag ← 1
	QNaN & src1 not a NaN	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← src2	v ← Q(src2) vxsnan_flag ← 1

Explanation:

src1	The double-precision floating-point value in doubleword element 0 of VSR[XA].
src2	For ***xsnmsubadp***, the double-precision floating-point value in doubleword element 0 of VSR[XT]. For ***xsnmsubmdp***, the double-precision floating-point value in doubleword element 0 of VSR[XB].
src3	For ***xsnmsubadp***, the double-precision floating-point value in doubleword element 0 of VSR[XB]. For ***xsnmsubmdp***, the double-precision floating-point value in doubleword element 0 of VSR[XT].
dQNaN	Default quiet NaN (`0x7FF8_0000_0000_0000`).
NZF	Nonzero finite number.
Rezd	Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs). Can also occur with two nonzero finite number source operands.
Q(x)	Return a QNaN with the payload of x.
S(x,y)	Return the normalized sum of floating-point value x and negated floating-point value y, having unbounded range and precision. Note: If x = y, v is considered to be an exact-zero-difference result (Rezd).
M(x,y)	Return the normalized product of floating-point value x and floating-point value y, having unbounded range and precision.
p	The intermediate product having unbounded range and precision.
v	The intermediate result having unbounded range and precision.

Table 7.35: Actions for xsnmsub(a|m)dp

VSX Scalar Negative Multiply-Subtract Type-A Single-Precision XX3-form

xsnmsubasp XT,XA,XB

60	T	A	B	145	AX	BX	TX
0	6	11	16	21	29	30	31

VSX Scalar Negative Multiply-Subtract Type-M Single-Precision XX3-form

xsnmsubmsp XT,XA,XB

60	T	A	B	153	AX	BX	TX
0	6	11	16	21	29	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

reset_xflags()

if ''xsnmsubasp'' then do
   src1 ← bfp_CONVERT_FROM_BFP64(VSR[32×AX+A].dword[0])
   src2 ← bfp_CONVERT_FROM_BFP64(VSR[32×TX+T].dword[0])
   src3 ← bfp_CONVERT_FROM_BFP64(VSR[32×BX+B].dword[0])
end
if ''xsnmsubmsp'' then do
   src1 ← bfp_CONVERT_FROM_BFP64(VSR[32×AX+A].dword[0])
   src2 ← bfp_CONVERT_FROM_BFP64(VSR[32×BX+B].dword[0])
   src3 ← bfp_CONVERT_FROM_BFP64(VSR[32×TX+T].dword[0])
end

v        ← bfp_MULTIPLY_ADD(src1, src3, bfp_NEGATE(src2))
rnd      ← bfp_NEGATE(bfp_ROUND_TO_BFP32(FPSCR.RN, v))
result32 ← bfp32_CONVERT_FROM_BFP(rnd)
result64 ← bfp64_CONVERT_FROM_BFP(rnd)

if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
if vximz_flag=1 then SetFX(FPSCR.VXIMZ)
if vxisi_flag=1 then SetFX(FPSCR.VXISI)
if ox_flag=1 then SetFX(FPSCR.OX)
if ux_flag=1 then SetFX(FPSCR.UX)
if xx_flag=1 then SetFX(FPSCR.XX)

vx_flag  ← vxsnan_flag | vximz_flag | vxisi_flag
vex_flag ← FPSCR.VE & vx_flag

if vex_flag=0 then do
   VSR[32×TX+T].dword[0] ← result64
   VSR[32×TX+T].dword[1] ← 0x0000_0000_0000_0000
   FPSCR.FPRF ← fprf_CLASS_BFP32(result32)
   FPSCR.FR   ← inc_flag
   FPSCR.FI   ← xx_flag
end
else do
   FPSCR.FR   ← 0b0
   FPSCR.FI   ← 0b0
end
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

For ***xsnmsubasp***, do the following.

- Let src1 be the double-precision floating-point value in doubleword element 0 of VSR[XA].
- Let src2 be the double-precision floating-point value in doubleword element 0 of VSR[XT].
- Let src3 be the double-precision floating-point value in doubleword element 0 of VSR[XB].

For ***xsnmsubmsp***, do the following.

- Let src1 be the double-precision floating-point value in doubleword element 0 of VSR[XA].
- Let src2 be the double-precision floating-point value in doubleword element 0 of VSR[XB].
- Let src3 be the double-precision floating-point value in doubleword element 0 of VSR[XT].

src1 is multiplied[61] by src3, producing a product having unbounded range and precision.

See part 1 of Table 7.36, “Actions for xsnmsub(a|m)sp,” on page 681.

src2 is negated and added[62] to the product, producing a sum having unbounded range and precision.

The sum is normalized[63].

See part 2 of Table 7.36, “Actions for xsnmsub(a|m)sp,” on page 681.

The intermediate result is rounded to single-precision using the rounding mode specified by RN.

See Table 7.9, “Scalar Floating-Point Intermediate Result Handling,” on page 617.

The result is negated and placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are set to 0.

FPRF is set to the class and sign of the result as represented in single-precision format. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.

If a trap-enabled invalid operation exception occurs, VSR[XT] and FPRF are not modified, and FR and FI are set to 0.

[61] Floating-point multiplication is based on exponent addition and multiplication of the significands.

[62] 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.

[63] Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.

See Table 7.32, “Scalar Floating-Point Final Result with Negation,” on page 669.

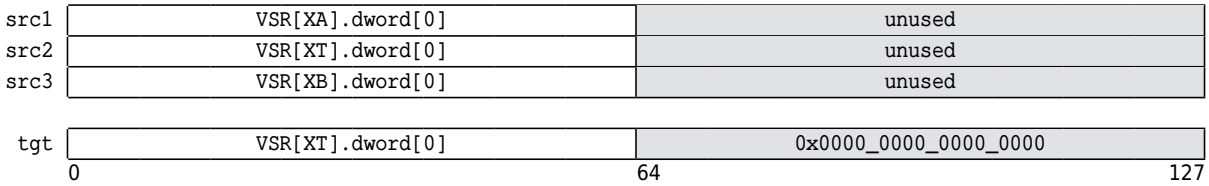
Special Registers Altered:

Register	*Field(s)*
FPSCR	FPRF FR FI FX OX UX XX VXSNAN VXISI VXIMZ

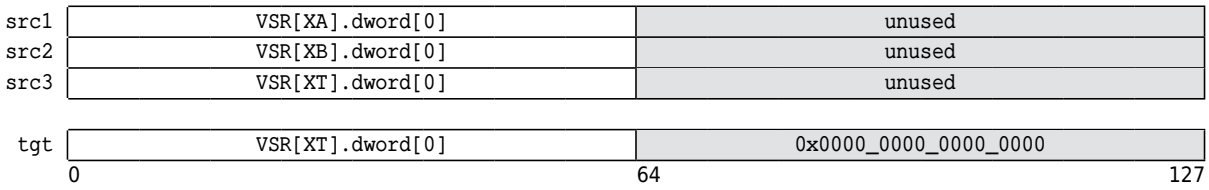
Programming Note

Previous versions of the architecture allowed the contents of doubleword 1 of the result register to be undefined. However, all processors that support this instruction write 0s into doubleword 1 of the result register, as is required by this version of the architecture.

VSR Data Layout for xsnmsubasp



VSR Data Layout for xsnmsubmsp



Part 1: Multiply	src3							
src1	−Infinity	−NZF	−Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
−Infinity	p ← +Infinity	p ← +Infinity	p ← dQNaN vximz_flag ← 1	p ← dQNaN vximz_flag ← 1	p ← −Infinity	p ← −Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
−NZF	p ← +Infinity	p ← M(src1,src3)	p ← +Zero	p ← −Zero	p ← M(src1,src3)	p ← −Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
−Zero	p ← dQNaN vximz_flag ← 1	p ← +Zero	p ← +Zero	p ← −Zero	p ← −Zero	p ← dQNaN vximz_flag ← 1	p ← src3	p ← Q(src3) vxsnan_flag ← 1
+Zero	p ← dQNaN vximz_flag ← 1	p ← −Zero	p ← −Zero	p ← +Zero	p ← +Zero	p ← dQNaN vximz_flag ← 1	p ← src3	p ← Q(src3) vxsnan_flag ← 1
+NZF	p ← −Infinity	p ← M(src1,src3)	p ← −Zero	p ← +Zero	p ← M(src1,src3)	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
+Infinity	p ← −Infinity	p ← −Infinity	p ← dQNaN vximz_flag ← 1	p ← dQNaN vximz_flag ← 1	p ← +Infinity	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
QNaN	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1 vxsnan_flag ← 1
SNaN	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1

Part 2: Subtract	src2							
p	−Infinity	−NZF	−Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
−Infinity	v ← dQNaN vxisi_flag ← 1	v ← −Infinity	v ← −Infinity	v ← −Infinity	v ← −Infinity	v ← −Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
−NZF	v ← +Infinity	v ← S(p,src2)	v ← p	v ← p	v ← S(p,src2)	v ← −Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
−Zero	v ← +Infinity	v ← −src2	v ← Rezd	v ← −Zero	v ← −src2	v ← −Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Zero	v ← +Infinity	v ← −src2	v ← +Zero	v ← Rezd	v ← −src2	v ← −Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+NZF	v ← +Infinity	v ← S(p,src2)	v ← p	v ← p	v ← S(p,src2)	v ← −Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← dQNaN vxisi_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
QNaN & src1 is a NaN	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p vxsnan_flag ← 1
QNaN & src1 not a NaN	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← src2	v ← Q(src2) vxsnan_flag ← 1

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]. For ***xsnmsubmsp***, the double-precision floating-point value in VSR[XB].dword[0].
src3	For ***xsnmsubasp***, the double-precision floating-point value in VSR[XB].dword[0]. For ***xsnmsubmsp***, the double-precision floating-point value in VSR[XT].dword[0].
dQNaN	Default quiet NaN (`0x7FF8_0000_0000_0000`).
NZF	Nonzero finite number.
Rezd	Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs). Can also occur with two nonzero finite number source operands.
Q(x)	Return a QNaN with the payload of x.
S(x,y)	Return the normalized sum of floating-point value x and negated floating-point value y, having unbounded range and precision. Note: If x = y, v is considered to be an exact-zero-difference result (Rezd).
M(x,y)	Return the normalized product of floating-point value x and floating-point value y, having unbounded range and precision.
p	The intermediate product having unbounded range and precision.
v	The intermediate result having unbounded range and precision.

Table 7.36: Actions for xsnmsub(a|m)sp

VSX Scalar Negative Multiply-Subtract Quad-Precision [using round to Odd] X-form

xsnmsubqp	VRT,VRA,VRB	(RO=0)
xsnmsubqpo	VRT,VRA,VRB	(RO=1)

63	VRT	VRA	VRB	484	RO
0	6	11	16	21	31

```
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 ← bfp128_CONVERT_FROM_BFP(rnd)

if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
if vximz_flag=1 then SetFX(FPSCR.VXIMZ)
if vxisi_flag=1 then SetFX(FPSCR.VXISI)
if ox_flag=1 then SetFX(FPSCR.OX)
if ux_flag=1 then SetFX(FPSCR.UX)
if xx_flag=1 then SetFX(FPSCR.XX)

vx_flag  ← vxsnan_flag | vximz_flag | vxisi_flag
ex_flag  ← FPSCR.VE & vx_flag

if ex_flag=0 then do
   VSR[VRT+32] ← result
   FPSCR.FPRF  ← fprf_CLASS_BFP128(result)
end
FPSCR.FR ← (vx_flag=0) & inc_flag
FPSCR.FI ← (vx_flag=0) & xx_flag
```

Let src1 be the floating-point value in VSR[VRA+32] represented in quad-precision format.

Let src2 be the floating-point value in VSR[VRT+32] represented in quad-precision format.

Let src3 be the floating-point value in VSR[VRB+32] represented in quad-precision format.

If either src1, src2, or src3 is a Signalling NaN, an Invalid Operation exception occurs and VXSNAN is set to 1.

If src1 is an Infinity value and src3 is a Zero value, or if src1 is a Zero value and src3 is an Infinity value, an Invalid Operation exception occurs and VXIMZ is set to 1.

If src2 and the product of src1 and src3 are Infinity values having same signs, an Invalid Operation exception occurs and VXISI is set to 1.

If src1 is a Signalling NaN, the result is the Quiet NaN corresponding to src1.

Otherwise, if src1 is a Quiet NaN, the result is src1.

Otherwise, if src2 is a Signalling NaN, the result is the Quiet NaN corresponding to src2.

Otherwise, if 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[64].

Otherwise, if the product of src1 and src3, and src2 are Infinity values having same signs, the result is the default Quiet NaN.

Otherwise, do the following.

> src1 is multiplied by src3, producing a product having unbounded significand precision and exponent range.
>
> See part 1 of Table 7.30. “Actions for xsmsubqp[o]”.
>
> src2 is negated and added to the product, producing a sum having unbounded range and precision.
>
> See part 2 of Table 7.30. “Actions for xsmsubqp[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 7.9, “Scalar Floating-Point Intermediate Result Handling,” on page 617.

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.

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.

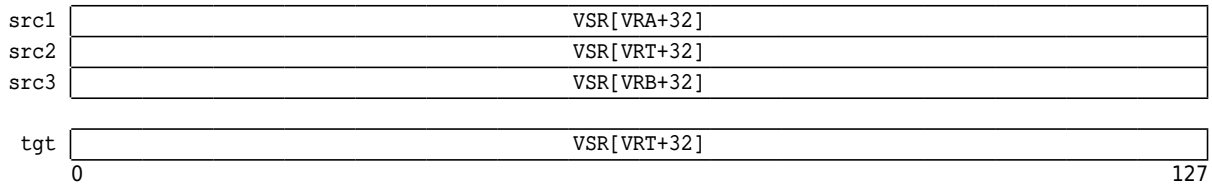
See Table 7.10, “VSX Scalar Floating-Point Final Result,” on page 618.

Special Registers Altered:

Register	*Field(s)*
FPSCR	FPRF FR FI FX VXSNAN VXIMZ VXISI OX UX XX

[64] The quad-precision default Quiet NaN is the value, 0x7FFF_8000_0000_0000_0000_0000_0000.

VSR Data Layout for xsnmsubqp[o]



Part 1: Multiply

src1 \ src3	−Infinity	−NZF	−Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
−Infinity	p ← +Infinity		p ← dQNaN vximz_flag ← 1			p ← −Infinity	p ← src3	p ← quiet(src3) vxsnan_flag ← 1
−NZF		p ← mul(src1,src3)			p ← mul(src1,src3)			
−Zero	p ← dQNaN vximz_flag ← 1	p ← +Zero		p ← −Zero		p ← dQNaN vximz_flag ← 1		
+Zero		p ← −Zero		p ← +Zero				
+NZF		p ← mul(src1,src3)			p ← mul(src1,src3)			
+Infinity	p ← −Infinity		p ← dQNaN vximz_flag ← 1			p ← +Infinity		
QNaN	p ← src1							p ← src1 vxsnan_flag ← 1
SNaN	p ← quiet(src1) vxsnan_flag ← 1							

Part 2: Subtract

p \ src2	−Infinity	−NZF	−Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
−Infinity	v ← dQNaN vxisi_flag ← 1					v ← −Infinity	v ← src2	v ← quiet(src2) vxsnan_flag ← 1
−NZF		v ← sub(p,src2)	v ← p		v ← sub(p,src2)			
−Zero		v ← −src2	v ← Rezd	v ← −Zero	v ← −src2			
+Zero			v ← +Zero	v ← Rezd				
+NZF		v ← sub(p,src2)	v ← p		v ← sub(p,src2)			
+Infinity	v ← +Infinity					v ← dQNaN vxisi_flag ← 1		
QNaN & src1 is a NaN	v ← p							v ← p vxsnan_flag ← 1
QNaN & src1 not a NaN							v ← src2	v ← quiet(src2) vxsnan_flag ← 1

Explanation:

src1	The quad-precision floating-point value in `VSR[VRA+32]`.
src2	The quad-precision floating-point value in `VSR[VRT+32]`.
src3	The quad-precision floating-point value in `VSR[VRB+32]`.
dQNaN	Default quiet NaN (`0x7FFF_8000_0000_0000_0000_0000_0000`).
NZF	Nonzero finite number.
Rezd	Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs). Can also occur with two nonzero finite number source operands.
quiet(x)	Return a QNaN with the payload of x.
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 7.37: Actions for xsnmsubqp[o]

VSX Scalar Reciprocal Estimate Double-Precision XX2-form

xsredp XT,XB

60	T	///	B	90	BX	TX
0	6	11	16	21	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

reset_xflags()

src    ← bfp_CONVERT_FROM_BFP64(VSR[VRB+32].dword[0])
v      ← bfp_RECIPROCAL_ESTIMATE(src)
rnd    ← bfp_ROUND_TO_BFP64(0b0,FPSCR.RN,v)
result ← bfp64_CONVERT_FROM_BFP(rnd)

if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
if ox_flag=1 then SetFX(FPSCR.OX)
if ux_flag=1 then SetFX(FPSCR.UX)
if zx_flag=1 then SetFX(FPSCR.ZX)

vex_flag ← FPSCR.VE & vxsnan_flag
zex_flag ← FPSCR.ZE & zx_flag

if vex_flag=0 & zex_flag=0 then do
   VSR[32×TX+T].dword[0] ← result
   VSR[32×TX+T].dword[1] ← 0x0000_0000_0000_0000
   FPSCR.FPRF ← fprf_CLASS_BFP64(result)
   FPSCR.FR   ← 0bU
   FPSCR.FI   ← 0bU
end
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

Let src be the double-precision floating-point value in doubleword element 0 of VSR[XB].

A double-precision floating-point estimate of the reciprocal of src is placed into doubleword element 0 of VSR[XT] in double-precision format.

Unless the reciprocal of src would be a zero, an infinity, or a QNaN, the estimate has a relative error in precision no greater than one part in 16384 of the reciprocal of src. That is,

$$\left| \frac{\text{estimate} - \frac{1}{\text{src}}}{\frac{1}{\text{src}}} \right| \leq \frac{1}{16384}$$

Operation with various special values of the operand is summarized below.

Source Value	Result	Exception
−Infinity	−Zero	None
−Zero	−Infinity[1]	ZX
+Zero	+Infinity[1]	ZX
+Infinity	+Zero	None
SNaN	QNaN[2]	VXSNAN
QNaN	QNaN	None

[1] No result if ZE=1.
[2] No result if VE=1.

The contents of doubleword element 1 of VSR[XT] are set to 0.

FPRF is set to the class and sign of the result. 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.

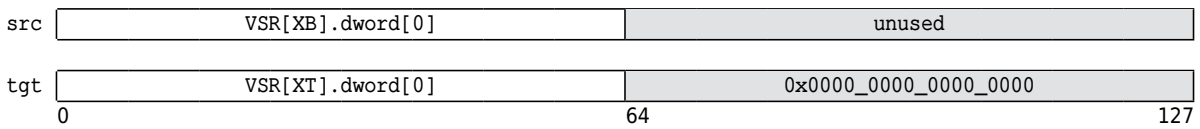
Special Registers Altered:

Register	*Field(s)*
FPSCR	FPRF FX OX UX VXSNAN
FPSCR	FR (undefined)
FPSCR	FI (undefined)
FPSCR	XX (undefined)

Programming Note

Previous versions of the architecture allowed the contents of doubleword 1 of the result register to be undefined. However, all processors that support this instruction write 0s into doubleword 1 of the result register, as is required by this version of the architecture.

VSR Data Layout for xsredp



VSX Scalar Reciprocal Estimate Single-Precision XX2-form

xsresp XT,XB

60	T	///	B	26	BX	TX
0	6	11	16	21	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

reset_xflags()

src      ← bfp_CONVERT_FROM_BFP64(VSR[VRB+32].dword[0])
v        ← bfp_RECIPROCAL_ESTIMATE(src)
rnd      ← bfp_ROUND_TO_BFP32(FPSCR.RN,v)
result32 ← bfp32_CONVERT_FROM_BFP(rnd)
result64 ← bfp64_CONVERT_FROM_BFP(rnd)

if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
if ox_flag=1 then SetFX(FPSCR.OX)
if ux_flag=1 then SetFX(FPSCR.UX)
if 0bU then SetFX(FPSCR.XX)
if zx_flag=1 then SetFX(FPSCR.ZX)

vex_flag ← FPSCR.VE & vxsnan_flag
zex_flag ← FPSCR.ZE & zx_flag

if vex_flag=0 & zex_flag=0 then do
   VSR[32×TX+T].dword[0] ← result64
   VSR[32×TX+T].dword[1] ← 0x0000_0000_0000_0000
   FPSCR.FPRF ← fprf_CLASS_BFP32(result32)
   FPSCR.FR   ← 0bU
   FPSCR.FI   ← 0bU
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].

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.

Source Value	Result	Exception
−Infinity	−Zero	None
−Zero	−Infinity[1]	ZX
+Zero	+Infinity[1]	ZX
+Infinity	+Zero	None
SNaN	QNaN[2]	VXSNAN
QNaN	QNaN	None

[1] No result if ZE=1.
[2] No result if VE=1.

The contents of doubleword element 1 of VSR[XT] are set to 0.

FPRF is set to the class and sign of the result as represented in single-precision format. FR is set to an undefined value. FI is set to an undefined value.

If a trap-enabled invalid operation exception or a trap-enabled zero divide exception occurs, VSR[XT] and FPRF are not modified.

The results of executing this instruction is permitted to vary between implementations, and between different executions on the same implementation.

Special Registers Altered:

Register	*Field(s)*
FPSCR	FPRF FX OX UX ZX VXSNAN
FPSCR	FR (undefined)
FPSCR	FI (undefined)
FPSCR	XX (undefined)

Programming Note

Previous versions of the architecture allowed the contents of doubleword 1 of the result register to be undefined. However, all processors that support this instruction write 0s into doubleword 1 of the result register, as is required by this version of the architecture.

VSR Data Layout for xsresp

VSX Scalar Reciprocal Square Root Estimate Double-Precision XX2-form

xsrsqrtedp XT,XB

60	T	///	B	74	BX	TX
0	6	11	16	21	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

reset_xflags()

src    ← bfp_CONVERT_FROM_BFP64(VSR[32×BX+B].dword[0])
v      ← bfp_RECIPROCAL_SQUARE_ROOT_ESTIMATE(src)
rnd    ← bfp_ROUND_TO_BFP64(0b0,FPSCR.RN,v)
result ← bfp64_CONVERT_FROM_BFP(rnd)

if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
if vxsqrt_flag=1 then SetFX(FPSCR.VXSQRT)
if zx_flag=1 then SetFX(FPSCR.ZX)

vx_flag  ← vxsnan_flag | vxsqrt_flag

vex_flag ← FPSCR.VE & vx_flag
zex_flag ← FPSCR.ZE & zx_flag

if vex_flag=0 & zex_flag=0 then do
   VSR[32×TX+T].dword[0] ← result
   VSR[32×TX+T].dword[1] ← 0x0000_0000_0000_0000
   FPSCR.FPRF ← fprf_CLASS_BFP64(result)
   FPSCR.FR   ← 0bU
   FPSCR.FI   ← 0bU
end
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

Let src be the double-precision floating-point value in doubleword element 0 of VSR[XB].

A double-precision floating-point estimate of the reciprocal square root of src is placed into doubleword element 0 of VSR[XT] in double-precision format.

Unless the reciprocal of the square root of src would be a zero, an infinity, or a QNaN, the estimate has a relative error in precision no greater than one part in 16384 of the reciprocal of the square root of src. That is,

$$\left| \frac{\text{estimate} - \frac{1}{\sqrt{\text{src}}}}{\frac{1}{\sqrt{\text{src}}}} \right| \leq \frac{1}{16384}$$

Operation with various special values of the operand is summarized below.

Source Value	Result	Exception
−Infinity	QNaN[1]	VXSQRT
−Finite	QNaN[1]	VXSQRT
−Zero	−Infinity[2]	ZX
+Zero	+Infinity[2]	ZX
+Infinity	+Zero	None
SNaN	QNaN[1]	VXSNAN
QNaN	QNaN	None

[1] No result if VE=1.
[2] No result if ZE=1.

The contents of doubleword element 1 of VSR[XT] are set to 0.

FPRF is set to the class and sign of the result. FR is set to an undefined value. FI is set to an undefined value.

If a trap-enabled invalid operation exception or a trap-enabled zero divide exception occurs, VSR[XT] and FPRF are not modified.

The results of executing this instruction is permitted to vary between implementations, and between different executions on the same implementation.

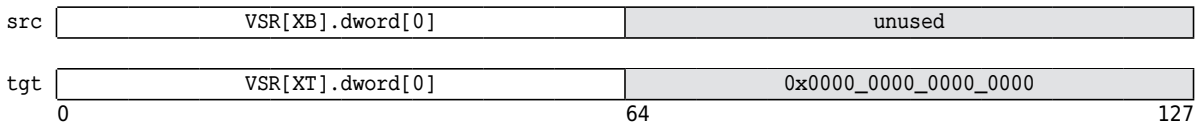
Special Registers Altered:

Register	*Field(s)*
FPSCR	FPRF FX VXSNAN VXSQRT
FPSCR	FR (undefined)
FPSCR	FI (undefined)
FPSCR	XX (undefined)

Programming Note

Previous versions of the architecture allowed the contents of doubleword 1 of the result register to be undefined. However, all processors that support this instruction write 0s into doubleword 1 of the result register, as is required by this version of the architecture.

VSR Data Layout for xsrsqrtedp



VSX Scalar Reciprocal Square Root Estimate Single-Precision XX2-form

xsrsqrtesp XT,XB

60	T	///	B	10	BX	TX
0	6	11	16	21	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

reset_xflags()

src  ← bfp_CONVERT_FROM_BFP64(VSR[32×BX+B].dword[0])
v    ← bfp_RECIPROCAL_SQUARE_ROOT_ESTIMATE(src)

rnd      ← bfp_ROUND_TO_BFP32(FPSCR.RN,v)
result32 ← bfp32_CONVERT_FROM_BFP(rnd)
result64 ← bfp64_CONVERT_FROM_BFP(rnd)

if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
if vxsqrt_flag=1 then SetFX(FPSCR.VXSQRT)
if ox_flag=1 then SetFX(FPSCR.OX)
if ux_flag=1 then SetFX(FPSCR.UX)
if 0bU then SetFX(FPSCR.XX)
if zx_flag=1 then SetFX(FPSCR.ZX)

vx_flag  ← vxsnan_flag | vxsqrt_flag

vex_flag ← FPSCR.VE & vx_flag
zex_flag ← FPSCR.ZE & zx_flag

if vex_flag=0 & zex_flag=0 then do
   VSR[32×TX+T].dword[0] ← result64
   VSR[32×TX+T].dword[1] ← 0x0000_0000_0000_0000
   FPSCR.FPRF ← fprf_CLASS_BFP32(result32)
   FPSCR.FR   ← 0bU
   FPSCR.FI   ← 0bU
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].

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,

$$\left| \frac{\text{estimate} - \frac{1}{\sqrt{\text{src}}}}{\frac{1}{\sqrt{\text{src}}}} \right| \leq \frac{1}{16384}$$

Operation with various special values of the operand is summarized below.

Source Value	Result	Exception
−Infinity	QNaN[1]	VXSQRT
−Finite	QNaN[1]	VXSQRT
−Zero	−Infinity[2]	ZX
+Zero	+Infinity[2]	ZX
+Infinity	+Zero	None
SNaN	QNaN[1]	VXSNAN
QNaN	QNaN	None

[1] No result if VE=1.
[2] No result if ZE=1.

The contents of doubleword element 1 of VSR[XT] are set to 0.

FPRF is set to the class and sign of the result as represented in single-precision format. FR is set to an undefined value. FI is set to an undefined value.

If a trap-enabled invalid operation exception or a trap-enabled zero divide exception occurs, VSR[XT] and FPRF are not modified.

The results of executing this instruction is permitted to vary between implementations, and between different executions on the same implementation.

Special Registers Altered:

Register	*Field(s)*
FPSCR	FPRF FX OX UX ZX VXSNAN VXSQRT
FPSCR	FR (undefined)
FPSCR	FI (undefined)
FPSCR	XX (undefined)

Programming Note

Previous versions of the architecture allowed the contents of doubleword 1 of the result register to be undefined. However, all processors that support this instruction write 0s into doubleword 1 of the result register, as is required by this version of the architecture.

VSR Data Layout for xsrsqrtesp

VSX Scalar Test for software Divide Double-Precision XX3-form

xstdivdp BF,XA,XB

60	BF	//	A	B	61	AX	BX	/
0	6	9	11	16	21	29	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

src1    ← VSR[32×AX+A].dword[0]
src2    ← VSR[32×BX+B].dword[0]
e_a     ← src1.bit[1:11] - 1023
e_b     ← src2.bit[1:11] - 1023
fe_flag ← IsNaN(src1) | IsInf(src1) |
          IsNaN(src2) | IsInf(src2) | IsZero(src2) |
          (e_b <= -1022) |
          (e_b >=  1021) |
          (!IsZero(src1) & ((e_a - e_b) >=  1023)) |
          (!IsZero(src1) & ((e_a - e_b) <= -1021)) |
          (!IsZero(src1) & (e_a <= -970))
fg_flag ← IsInf(src1) | IsInf(src2) |
          IsZero(src2) | IsDen(src2)
fl_flag ← xsredp_error() <= 2^-14
CR[BF]  ← 0b1 || fg_flag || fe_flag || 0b0
```

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].

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 field BF is set to the value 0b1 || fg_flag || fe_flag || 0b0.

Special Registers Altered:

Register	*Field(s)*
CR	field BF

VSR Data Layout for xstdivdp

VSX Scalar Test for software Square Root Double-Precision XX2-form

xstsqrtdp BF,XB

60	BF	//	///	B	106	BX	/
0	6	9	11	16	21	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

src     ← VSR[32×BX+B].dword[0]
e_b     ← src.bit[1:11] - 1023
fe_flag ← IsNaN(src) | IsInf(src) | IsZero(src) |
          IsNeg(src) | (e_b <= -970)
fg_flag ← IsInf(src) | IsZero(src) | IsDen(src)
fl_flag ← xsrsqrtedp_error() <= 2^-14

CR.field[BF] ← 0b1 || fg_flag || fe_flag || 0b0
```

Let XB be the value 32×BX + B.

Let src be the double-precision floating-point value in doubleword element 0 of VSR[XB].

Let 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:

Register	*Field(s)*
CR	field BF

VSR Data Layout for xstsqrtdp

7.6.2.3.2 VSX Vector BFP Arithmetic Instructions

VSX Vector Add Double-Precision XX3-form

xvadddp XT,XA,XB

60	T	A	B	96	AX	BX	TX
0	6	11	16	21	29	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

ex_flag ← 0b0

do i = 0 to 1
   reset_xflags()

   src1 ← bfp_CONVERT_FROM_BFP64(VSR[32×AX+A].dword[i])
   src2 ← bfp_CONVERT_FROM_BFP64(VSR[32×BX+B].dword[i])
   v    ← bfp_ADD(src1,src2)
   rnd  ← bfp_ROUND_TO_BFP64(0b0,FPSCR.RN,v)

   vresult.dword[i] ← bfp64_CONVERT_FROM_BFP(rnd)

   if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
   if vxisi_flag=1 then SetFX(FPSCR.VXISI)
   if ox_flag=1 then SetFX(FPSCR.OX)
   if ux_flag=1 then SetFX(FPSCR.UX)
   if xx_flag=1 then SetFX(FPSCR.XX)

   ex_flag ← ex_flag | (FPSCR.VE & vxsnan_flag)
                     | (FPSCR.VE & vxisi_flag)
                     | (FPSCR.OE & ox_flag)
                     | (FPSCR.UE & ux_flag)
                     | (FPSCR.XE & xx_flag)
end

if ex_flag=0 then VSR[32×TX+T] ← vresult
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

For each integer value i from 0 to 1, do the following.

Let src1 be the double-precision floating-point operand in doubleword element i of VSR[XA].

Let src2 be the double-precision floating-point operand in doubleword element i of VSR[XB].

src2 is added[65] to src1, producing a sum having unbounded range and precision.

The sum is normalized[66].

See Table 7.38.

The intermediate result is rounded to double-precision using the rounding mode specified by RN.

See Table 7.9, “Scalar Floating-Point Intermediate Result Handling,” on page 617.

The result is placed into doubleword element i of VSR[XT] in double-precision format.

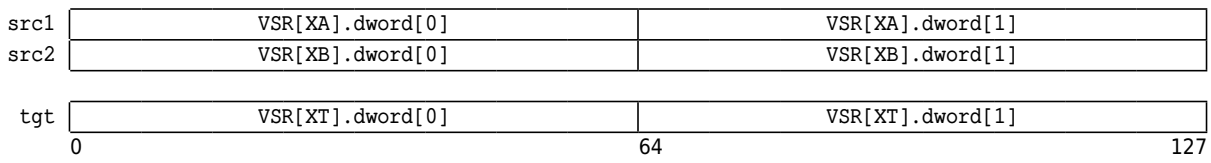
See Table 7.39, “Vector Floating-Point Final Result,” on page 693.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered:

Register	*Field(s)*
FPSCR	FX OX UX XX VXSNAN VXISI

VSR Data Layout for xvadddp



[65] 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.

[66] 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.

src1 \ src2	−Infinity	−NZF	−Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
−Infinity	v ← −Infinity	v ← −Infinity	v ← −Infinity	v ← −Infinity	v ← −Infinity	v ← dQNaN vxisi_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
−NZF	v ← −Infinity	v ← A(src1,src2)	v ← src1	v ← src1	v ← A(src1,src2)	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
−Zero	v ← −Infinity	v ← src2	v ← −Zero	v ← Rezd	v ← src2	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Zero	v ← −Infinity	v ← src2	v ← Rezd	v ← +Zero	v ← src2	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+NZF	v ← −Infinity	v ← A(src1,src2)	v ← src1	v ← src1	v ← A(src1,src2)	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Infinity	v ← dQNaN vxisi_flag ← 1	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
QNaN	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1 vxsnan_flag ← 1
SNaN	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

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 signficand precision and unbounded exponent range.

Table 7.38: Actions for xvadddp (element i)

Case	FPSCR.VE	FPSCR.OE	FPSCR.UE	FPSCR.ZE	FPSCR.XE	vxsnan_flag	vximz_flag	vxisi_flag	vxidi_flag	vxzdz_flag	vxsqrt_flag	zx_flag	Is r inexact? (r ≠ v)	Is r incremented? (|r| > |v|)	Is q inexact? (q ≠ v)	Is q incremented? (|q| > |v|)	Returned Results and Status Setting
Special	–	–	–	–	–	0	0	0	0	0	0	0	–	–	–	–	T(r)
	–	–	–	0	–	–	–	–	–	–	–	1	–	–	–	–	T(r), fx(ZX)
	–	–	–	1	–	–	–	–	–	–	–	1	–	–	–	–	fx(ZX), error()
	0	–	–	–	–	–	–	–	–	–	1	–	–	–	–	–	T(r), fx(VXSQRT)
	0	–	–	–	–	–	–	–	–	1	–	–	–	–	–	–	T(r), fx(VXZDZ)
	0	–	–	–	–	–	–	–	1	–	–	–	–	–	–	–	T(r), fx(VXIDI)
	0	–	–	–	–	–	–	1	–	–	–	–	–	–	–	–	T(r), fx(VXISI)
	0	–	–	–	–	0	1	–	–	–	–	–	–	–	–	–	T(r), fx(VXIMZ)
	0	–	–	–	–	1	0	–	–	–	–	–	–	–	–	–	T(r), fx(VXSNAN)
	0	–	–	–	–	1	1	–	–	–	–	–	–	–	–	–	T(r), fx(VXSNAN), fx(VXIMZ)
	1	–	–	–	–	–	–	–	–	–	1	–	–	–	–	–	T(r), fx(VXSQRT)
	1	–	–	–	–	–	–	–	–	1	–	–	–	–	–	–	fx(VXZDZ), error()
	1	–	–	–	–	–	–	–	1	–	–	–	–	–	–	–	fx(VXIDI), error()
	1	–	–	–	–	–	–	1	–	–	–	–	–	–	–	–	fx(VXISI), error()
	1	–	–	–	–	0	1	–	–	–	–	–	–	–	–	–	fx(VXIMZ), error()
	1	–	–	–	–	1	0	–	–	–	–	–	–	–	–	–	fx(VXSNAN), error()
	1	–	–	–	–	1	1	–	–	–	–	–	–	–	–	–	fx(VXSNAN), fx(VXIMZ), error()
Normal	–	–	–	–	–	–	–	–	–	–	–	–	no	–	–	–	T(r)
	–	–	–	–	0	–	–	–	–	–	–	–	yes	no	–	–	T(r), fx(XX)
	–	–	–	–	0	–	–	–	–	–	–	–	yes	yes	–	–	T(r), fx(XX)
	–	–	–	–	1	–	–	–	–	–	–	–	yes	no	–	–	T(r), fx(XX), error()
	–	–	–	–	1	–	–	–	–	–	–	–	yes	yes	–	–	T(r), fx(XX), error()
Overflow	–	0	–	–	0	–	–	–	–	–	–	–	–	–	–	–	T(r), fx(OX), fx(XX)
	–	0	–	–	1	–	–	–	–	–	–	–	–	–	–	–	T(r), fx(OX), fx(XX), error()
	–	1	–	–	–	–	–	–	–	–	–	–	–	–	no	–	fx(OX), error()
	–	1	–	–	–	–	–	–	–	–	–	–	–	–	yes	no	fx(OX), fx(XX), error()
	–	1	–	–	–	–	–	–	–	–	–	–	–	–	yes	yes	fx(OX), fx(XX), error()
Tiny	–	–	0	–	–	–	–	–	–	–	–	–	no	–	–	–	T(r)
	–	–	0	–	0	–	–	–	–	–	–	–	yes	no	–	–	T(r), fx(UX), fx(XX)
	–	–	0	–	0	–	–	–	–	–	–	–	yes	yes	–	–	T(r), fx(UX), fx(XX)
	–	–	0	–	1	–	–	–	–	–	–	–	yes	no	–	–	T(r), fx(UX), fx(XX), error()
	–	–	0	–	1	–	–	–	–	–	–	–	yes	yes	–	–	T(r), fx(UX), fx(XX), error()
	–	–	1	–	–	–	–	–	–	–	–	–	yes	–	no	–	fx(UX), error()
	–	–	1	–	–	–	–	–	–	–	–	–	yes	–	yes	no	fx(UX), fx(XX), error()
	–	–	1	–	–	–	–	–	–	–	–	–	yes	–	yes	yes	fx(UX), fx(XX), error()

Explanation:

–	The results do not depend on this condition.
fx(x)	FPSCR.FX is set to 1 if FPSCR.x=0. FPSCR.x is set to 1.
q	The value defined in Table 7.9, “Scalar Floating-Point Intermediate Result Handling,” on page 617, signficand rounded to the target precision, unbounded exponent range.
r	The value defined in Table 7.9, “Scalar Floating-Point Intermediate Result Handling,” on page 617, signficand rounded to the target precision, bounded exponent range.
v	The precise intermediate result defined in the instruction having unbounded signficand precision, unbounded exponent range.
error()	The system error handler is invoked for the trap-enabled exception if MSR.FE0 and MSR.FE1 bits are set to any mode other than the ignore-exception mode. Update of the target VSR is suppressed for all vector elements.
T(x)	The value x is placed in element i of VSR[XT] in the target precision format (where i∈{0,1} for results with 64-bit elements, and i∈{0,1,2,3} for results with 32-bit elements).

Table 7.39: Vector Floating-Point Final Result

VSX Vector Add Single-Precision XX3-form

xvaddsp XT,XA,XB

60	T	A	B	64	AX	BX	TX
0	6	11	16	21	29	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

ex_flag ← 0b0

do i = 0 to 3
   reset_xflags()

   src1 ← bfp_CONVERT_FROM_BFP32(VSR[32×AX+A].word[i])
   src2 ← bfp_CONVERT_FROM_BFP32(VSR[32×BX+B].word[i])
   v    ← bfp_ADD(src1,src2)
   rnd  ← bfp_ROUND_TO_BFP32(FPSCR.RN,v)

   vresult.word[i] ← bfp32_CONVERT_FROM_BFP(rnd)

   if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
   if vxisi_flag=1 then SetFX(FPSCR.VXISI)
   if ox_flag=1 then SetFX(FPSCR.OX)
   if ux_flag=1 then SetFX(FPSCR.UX)
   if xx_flag=1 then SetFX(FPSCR.XX)

   ex_flag ← ex_flag | (FPSCR.VE & vxsnan_flag)
                     | (FPSCR.VE & vxisi_flag)
                     | (FPSCR.OE & ox_flag)
                     | (FPSCR.UE & ux_flag)
                     | (FPSCR.XE & xx_flag)
end

if ex_flag=0 then VSR[32×TX+T] ← vresult
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

For each integer value i from 0 to 3, do the following.

Let src1 be the single-precision floating-point operand in word element i of VSR[XA].

Let src2 be the single-precision floating-point operand in word element i of VSR[XB].

src2 is added[67] to src1, producing a sum having unbounded range and precision.

The sum is normalized[68].

See Table 7.40.

The intermediate result is rounded to single-precision using the rounding mode specified by RN.

See Table 7.9, “Scalar Floating-Point Intermediate Result Handling,” on page 617.

The result is placed into word element i of VSR[XT] in single-precision format.

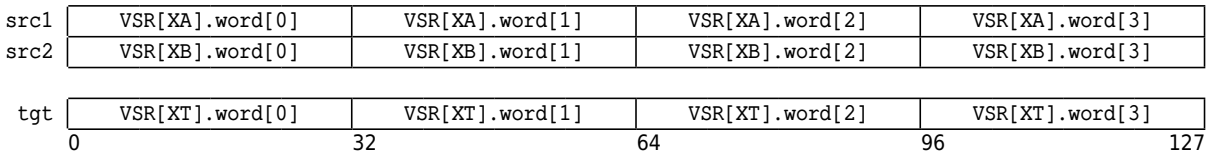
See Table 7.39, “Vector Floating-Point Final Result,” on page 693.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered:

Register	*Field(s)*
FPSCR	FX OX UX XX VXSNAN VXISI

VSR Data Layout for xvaddsp



[67] 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.

[68] 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.

		src2							
		−Infinity	**−NZF**	**−Zero**	**+Zero**	**+NZF**	**+Infinity**	**QNaN**	**SNaN**
src1	**−Infinity**	v ← −Infinity	v ← −Infinity	v ← −Infinity	v ← −Infinity	v ← −Infinity	v ← dQNaN vxisi_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	−NZF	v ← −Infinity	v ← A(src1,src2)	v ← src1	v ← src1	v ← A(src1,src2)	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	−Zero	v ← −Infinity	v ← src2	v ← −Zero	v ← Rezd	v ← src2	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+Zero	v ← −Infinity	v ← src2	v ← Rezd	v ← +Zero	v ← src2	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+NZF	v ← −Infinity	v ← A(src1,src2)	v ← src1	v ← src1	v ← A(src1,src2)	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+Infinity	v ← dQNaN vxisi_flag ← 1	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	QNaN	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1 vxsnan_flag ← 1
	SNaN	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

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 signficand precision and unbounded exponent range.

Table 7.40: Actions for xvaddsp (element i)

VSX Vector Divide Double-Precision XX3-form

xvdivdp XT,XA,XB

60	T	A	B	120	AX	BX	TX
0	6	11	16	21	29	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

ex_flag ← 0b0

do i = 0 to 1
   reset_xflags()

   src1 ← bfp_CONVERT_FROM_BFP64(VSR[32×AX+A].dword[i])
   src2 ← bfp_CONVERT_FROM_BFP64(VSR[32×BX+B].dword[i])
   v    ← bfp_DIVIDE(src1,src2)
   rnd  ← bfp_ROUND_TO_BFP64(0b0,FPSCR.RN,v)

   vresult.dword[i] ← bfp64_CONVERT_FROM_BFP(rnd)

   if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
   if vxidi_flag=1 then SetFX(FPSCR.VXIDI)
   if vxisi_flag=1 then SetFX(FPSCR.VXZDZ)
   if ox_flag=1 then SetFX(FPSCR.OX)
   if ux_flag=1 then SetFX(FPSCR.UX)
   if xx_flag=1 then SetFX(FPSCR.XX)
   if zx_flag=1 then SetFX(FPSCR.ZX)

   ex_flag ← ex_flag | (FPSCR.VE & vxsnan_flag)
                     | (FPSCR.VE & vxidi_flag)
                     | (FPSCR.VE & vxzdz_flag)
                     | (FPSCR.OE & ox_flag)
                     | (FPSCR.UE & ux_flag)
                     | (FPSCR.ZE & zx_flag)
                     | (FPSCR.XE & xx_flag)
end

if ex_flag=0 then VSR[32×TX+T] ← vresult
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

For each integer value i from 0 to 1, do the following.

Let src1 be the double-precision floating-point operand in doubleword element i of VSR[XA].
Let src2 be the double-precision floating-point operand in doubleword element i of VSR[XB].

src1 is divided[69] by src2, producing a quotient having unbounded range and precision.

The quotient is normalized[70].

See Table 7.41.

The intermediate result is rounded to double-precision using the rounding mode specified by RN.

See Table 7.9, “Scalar Floating-Point Intermediate Result Handling,” on page 617.

The result is placed into doubleword element i of VSR[XT] in double-precision format.

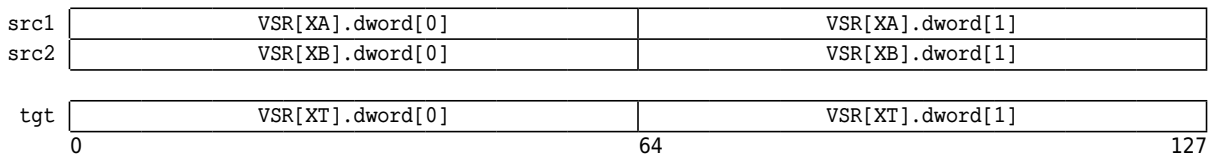
See Table 7.39, “Vector Floating-Point Final Result,” on page 693.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered:

Register	*Field(s)*
FPSCR	FX OX UX ZX XX VXSNAN VXIDI VXZDZ

VSR Data Layout for xvdivdp



[69] Floating-point division is based on exponent subtraction and division of the significands.
[70] 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.

		src2							
		−Infinity	**−NZF**	**−Zero**	**+Zero**	**+NZF**	**+Infinity**	**QNaN**	**SNaN**
src1	**−Infinity**	v ← dQNaN vxidi_flag ← 1	v ← +Infinity	v ← +Infinity	v ← −Infinity	v ← −Infinity	v ← dQNaN vxidi_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	−NZF	v ← +Zero	v ← D(src1,src2)	v ← +Infinity zx_flag ← 1	v ← −Infinity zx_flag ← 1	v ← D(src1,src2)	v ← −Zero	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	−Zero	v ← +Zero	v ← +Zero	v ← dQNaN vxzdz_flag ← 1	v ← dQNaN vxzdz_flag ← 1	v ← −Zero	v ← −Zero	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+Zero	v ← −Zero	v ← −Zero	v ← dQNaN vxzdz_flag ← 1	v ← dQNaN vxzdz_flag ← 1	v ← +Zero	v ← +Zero	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+NZF	v ← −Zero	v ← D(src1,src2)	v ← −Infinity zx_flag ← 1	v ← +Infinity zx_flag ← 1	v ← D(src1,src2)	v ← +Zero	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+Infinity	v ← dQNaN vxidi_flag ← 1	v ← −Infinity	v ← −Infinity	v ← +Infinity	v ← +Infinity	v ← dQNaN vxidi_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	QNaN	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1 vxsnan_flag ← 1
	SNaN	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

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.
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 signficand precision and unbounded exponent range.

Table 7.41: Actions for xvdivdp (element i)

VSX Vector Divide Single-Precision XX3-form

xvdivsp XT,XA,XB

60	T	A	B	88	AX	BX	TX
0	6	11	16	21	29	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

ex_flag ← 0b0

do i = 0 to 3
   reset_xflags()

   src1 ← bfp_CONVERT_FROM_BFP32(VSR[32×AX+A].word[i])
   src2 ← bfp_CONVERT_FROM_BFP32(VSR[32×BX+B].word[i])
   v    ← bfp_DIVIDE(src1,src2)
   rnd  ← bfp_ROUND_TO_BFP32(FPSCR.RN,v)

   vresult.word[i] ← bfp32_CONVERT_FROM_BFP(rnd)

   if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
   if vxidi_flag=1 then SetFX(FPSCR.VXIDI)
   if vxisi_flag=1 then SetFX(FPSCR.VXZDZ)
   if ox_flag=1 then SetFX(FPSCR.OX)
   if ux_flag=1 then SetFX(FPSCR.UX)
   if xx_flag=1 then SetFX(FPSCR.XX)
   if zx_flag=1 then SetFX(FPSCR.ZX)

   ex_flag ← ex_flag | (FPSCR.VE & vxsnan_flag)
                     | (FPSCR.VE & vxidi_flag)
                     | (FPSCR.VE & vxzdz_flag)
                     | (FPSCR.OE & ox_flag)
                     | (FPSCR.UE & ux_flag)
                     | (FPSCR.ZE & zx_flag)
                     | (FPSCR.XE & xx_flag)
end

if ex_flag=0 then VSR[32×TX+T] ← vresult
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

For each integer value i from 0 to 3, do the following.

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[71] by src2, producing a quotient having unbounded range and precision.

The quotient is normalized[72].

See Table 7.42.

The intermediate result is rounded to single-precision using the rounding mode specified by RN.

See Table 7.9, “Scalar Floating-Point Intermediate Result Handling,” on page 617.

The result is placed into word element i of VSR[XT] in single-precision format.

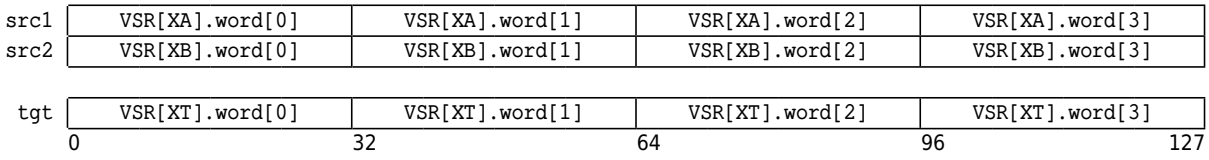
See Table 7.39, “Vector Floating-Point Final Result,” on page 693.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered:

Register	*Field(s)*
FPSCR	FX OX UX ZX XX VXSNAN VXIDI VXZDZ

VSR Data Layout for xvdivsp



[71] Floating-point division is based on exponent subtraction and division of the significands.
[72] 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.

		src2							
		−Infinity	**−NZF**	**−Zero**	**+Zero**	**+NZF**	**+Infinity**	**QNaN**	**SNaN**
src1	**−Infinity**	v ← dQNaN vxidi_flag ← 1	v ← +Infinity	v ← +Infinity	v ← −Infinity	v ← −Infinity	v ← dQNaN vxidi_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	−NZF	v ← +Zero	v ← D(src1,src2)	v ← +Infinity zx_flag ← 1	v ← −Infinity zx_flag ← 1	v ← D(src1,src2)	v ← −Zero	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	−Zero	v ← +Zero	v ← +Zero	v ← dQNaN vxzdz_flag ← 1	v ← dQNaN vxzdz_flag ← 1	v ← −Zero	v ← −Zero	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+Zero	v ← −Zero	v ← −Zero	v ← dQNaN vxzdz_flag ← 1	v ← dQNaN vxzdz_flag ← 1	v ← +Zero	v ← +Zero	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+NZF	v ← −Zero	v ← D(src1,src2)	v ← −Infinity zx_flag ← 1	v ← +Infinity zx_flag ← 1	v ← D(src1,src2)	v ← +Zero	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+Infinity	v ← dQNaN vxidi_flag ← 1	v ← −Infinity	v ← −Infinity	v ← +Infinity	v ← +Infinity	v ← dQNaN vxidi_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	QNaN	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1 vxsnan_flag ← 1
	SNaN	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

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.
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 signficand precision and unbounded exponent range.

Table 7.42: Actions for xvdivsp (element i)

VSX Vector Multiply Double-Precision XX3-form

xvmuldp XT,XA,XB

60	T	A	B	112	AX	BX	TX
0	6	11	16	21	29	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

ex_flag ← 0b0

do i = 0 to 1
   reset_xflags()

   src1 ← bfp_CONVERT_FROM_BFP64(VSR[32×AX+A].dword[i])
   src3 ← bfp_CONVERT_FROM_BFP64(VSR[32×BX+B].dword[i])
   v    ← bfp_MULTIPLY(src1,src3)
   rnd  ← bfp_ROUND_TO_BFP64(0b0,FPSCR.RN,v)

   vresult.dword[i] ← bfp64_CONVERT_FROM_BFP(rnd)

   if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
   if vximz_flag=1 then SetFX(FPSCR.VXIMZ)
   if ox_flag=1 then SetFX(FPSCR.OX)
   if ux_flag=1 then SetFX(FPSCR.UX)
   if xx_flag=1 then SetFX(FPSCR.XX)

   ex_flag ← ex_flag | (FPSCR.VE & vxsnan_flag)
                     | (FPSCR.VE & vximz_flag)
                     | (FPSCR.OE & ox_flag)
                     | (FPSCR.UE & ux_flag)
                     | (FPSCR.XE & xx_flag)
end

if ex_flag=0 then VSR[32×TX+T] ← vresult
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

For each integer value i from 0 to 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[73] by src2, producing a product having unbounded range and precision.

The product is normalized[74].

See Table 7.43.

The intermediate result is rounded to double-precision using the rounding mode specified by RN.

See Table 7.9, “Scalar Floating-Point Intermediate Result Handling,” on page 617.

The result is placed into doubleword element i of VSR[XT] in double-precision format.

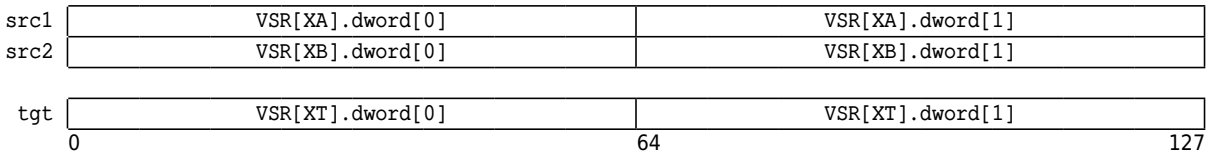
See Table 7.39, “Vector Floating-Point Final Result,” on page 693.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered:

Register	*Field(s)*
FPSCR	FX OX UX XX VXSNAN VXIMZ

VSR Data Layout for xvmuldp



[73] Floating-point multiplication is based on exponent addition and multiplication of the significands.
[74] 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.

		src2							
		−Infinity	**−NZF**	**−Zero**	**+Zero**	**+NZF**	**+Infinity**	**QNaN**	**SNaN**
src1	**−Infinity**	v ← +Infinity	v ← +Infinity	v ← dQNaN vximz_flag ← 1	v ← dQNaN vximz_flag ← 1	v ← −Infinity	v ← −Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	−NZF	v ← +Infinity	v ← M(src1,src2)	v ← +Zero	v ← −Zero	v ← M(src1,src2)	v ← −Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	−Zero	v ← dQNaN vximz_flag ← 1	v ← +Zero	v ← +Zero	v ← −Zero	v ← −Zero	v ← dQNaN vximz_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+Zero	v ← dQNaN vximz_flag ← 1	v ← −Zero	v ← −Zero	v ← +Zero	v ← +Zero	v ← dQNaN vximz_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+NZF	v ← −Infinity	v ← M(src1,src2)	v ← −Zero	v ← +Zero	v ← M(src1,src2)	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+Infinity	v ← −Infinity	v ← −Infinity	v ← dQNaN vximz_flag ← 1	v ← dQNaN vximz_flag ← 1	v ← +Infinity	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	QNaN	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1 vxsnan_flag ← 1
	SNaN	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

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.
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 signficand precision and unbounded exponent range.

Table 7.43: Actions for xvmuldp

VSX Vector Multiply Single-Precision XX3-form

xvmulsp XT,XA,XB

60	T	A	B	80	AX	BX	TX
0	6	11	16	21	29	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

ex_flag ← 0b0

do i = 0 to 3
   reset_xflags()

   src1 ← bfp_CONVERT_FROM_BFP32(VSR[32×AX+A].word[i])
   src3 ← bfp_CONVERT_FROM_BFP32(VSR[32×BX+B].word[i])
   v    ← bfp_MULTIPLY(src1,src3)
   rnd  ← bfp_ROUND_TO_BFP32(FPSCR.RN,v)

   vresult.word[i] ← bfp32_CONVERT_FROM_BFP(rnd)

   if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
   if vximz_flag=1 then SetFX(FPSCR.VXIMZ)
   if ox_flag=1 then SetFX(FPSCR.OX)
   if ux_flag=1 then SetFX(FPSCR.UX)
   if xx_flag=1 then SetFX(FPSCR.XX)

   ex_flag ← ex_flag | (FPSCR.VE & vxsnan_flag)
                     | (FPSCR.VE & vximz_flag)
                     | (FPSCR.OE & ox_flag)
                     | (FPSCR.UE & ux_flag)
                     | (FPSCR.XE & xx_flag)
end

if ex_flag=0 then VSR[32×TX+T] ← vresult
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

For each integer value i from 0 to 3, do the following.

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[75] by src2, producing a product having unbounded range and precision.

The product is normalized[76].

See Table 7.44.

The intermediate result is rounded to single-precision using the rounding mode specified by RN.

See Table 7.9, “Scalar Floating-Point Intermediate Result Handling,” on page 617.

The result is placed into word element i of VSR[XT] in single-precision format.

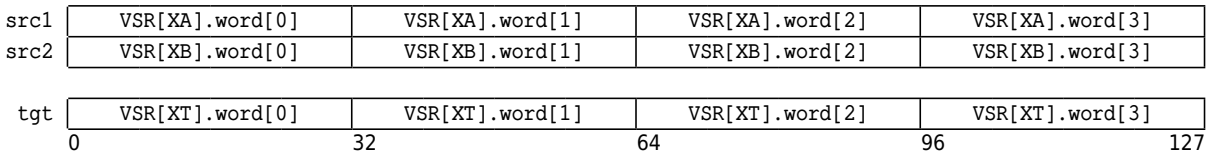
See Table 7.39, “Vector Floating-Point Final Result,” on page 693.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered:

Register	*Field(s)*
FPSCR	FX OX UX XX VXSNAN VXIMZ

VSR Data Layout for xvmulsp



[75] Floating-point multiplication is based on exponent addition and multiplication of the significands.
[76] 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.

		src2							
		−Infinity	**−NZF**	**−Zero**	**+Zero**	**+NZF**	**+Infinity**	**QNaN**	**SNaN**
src1	**−Infinity**	v ← +Infinity	v ← +Infinity	v ← dQNaN vximz_flag ← 1	v ← dQNaN vximz_flag ← 1	v ← −Infinity	v ← −Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	−NZF	v ← +Infinity	v ← M(src1,src2)	v ← +Zero	v ← −Zero	v ← M(src1,src2)	v ← −Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	−Zero	v ← dQNaN vximz_flag ← 1	v ← +Zero	v ← +Zero	v ← −Zero	v ← −Zero	v ← dQNaN vximz_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+Zero	v ← dQNaN vximz_flag ← 1	v ← −Zero	v ← −Zero	v ← +Zero	v ← +Zero	v ← dQNaN vximz_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+NZF	v ← −Infinity	v ← M(src1,src2)	v ← −Zero	v ← +Zero	v ← M(src1,src2)	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+Infinity	v ← −Infinity	v ← −Infinity	v ← dQNaN vximz_flag ← 1	v ← dQNaN vximz_flag ← 1	v ← +Infinity	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	QNaN	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1 vxsnan_flag ← 1
	SNaN	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

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 signficand precision and unbounded exponent range.

Table 7.44: Actions for xvmulsp

VSX Vector Square Root Double-Precision XX2-form

xvsqrtdp XT,XB

60	T	///	B	203	BX	TX
0	6	11	16	21	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

ex_flag ← 0b0

do i = 0 to 1
   reset_xflags()

   src ← bfp_CONVERT_FROM_BFP64(VSR[32×BX+B].dword[i])
   v   ← bfp_SQUARE_ROOT(src)
   rnd ← bfp_ROUND_TO_BFP64(0b0,FPSCR.RN,v)

   vresult.dword[i] ← bfp64_CONVERT_FROM_BFP(rnd)

   if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
   if vxsqrt_flag=1 then SetFX(FPSCR.VXSQRT)
   if xx_flag=1 then SetFX(FPSCR.XX)

   ex_flag ← ex_flag | (FPSCR.VE & vxsnan_flag)
                     | (FPSCR.VE & vxsqrt_flag)
                     | (FPSCR.XE & xx_flag)
end

if ex_flag=0 then VSR[32×TX+T] ← vresult
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

For each integer value i from 0 to 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 7.45.

The intermediate result is rounded to double-precision using the rounding mode specified by RN.

See Table 7.9, “Scalar Floating-Point Intermediate Result Handling,” on page 617.

The result is placed into doubleword element i of VSR[XT] in double-precision format.

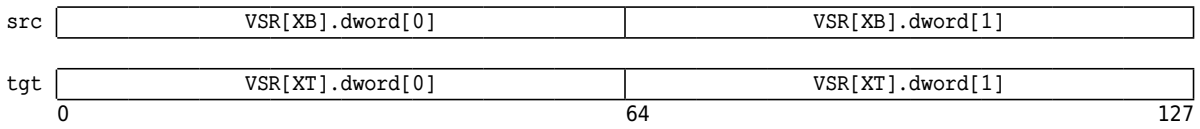
See Table 7.39, “Vector Floating-Point Final Result,” on page 693.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered:

Register	*Field(s)*
FPSCR	FX XX VXSNAN VXSQRT

VSR Data Layout for xvsqrtdp



src							
−Infinity	**−NZF**	**−Zero**	**+Zero**	**+NZF**	**+Infinity**	**QNaN**	**SNaN**
v ← dQNaN vxsqrt_flag ← 1	v ← dQNaN vxsqrt_flag ← 1	v ← +Zero	v ← +Zero	v ← SQRT(src)	v ← +Infinity	v ← src	v ← Q(src) vxsnan_flag ← 1

Explanation:

src	The double-precision floating-point value in doubleword element i of VSR[XB] (where i∈{0,1}).
dQNaN	Default quiet NaN (0x7FF8_0000_0000_0000).
NZF	Nonzero finite number.
SQRT(x)	The unbounded-precision square root of the floating-point value x.
Q(x)	Return a QNaN with the payload of x.
v	The intermediate result having unbounded signficand precision and unbounded exponent range.

Table 7.45: Actions for xvsqrtdp

VSX Vector Square Root Single-Precision XX2-form

xvsqrtsp XT,XB

60	T	///	B	139	BX	TX
0	6	11	16	21	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

ex_flag ← 0b0

do i = 0 to 3
   reset_xflags()

   src ← bfp_CONVERT_FROM_BFP64(VSR[32×BX+B].word[i])
   v   ← bfp_SQUARE_ROOT(src)
   rnd ← bfp_ROUND_TO_BFP32(FPSCR.RN,v)

   vresult.word[i] ← bfp32_CONVERT_FROM_BFP(rnd)

   if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
   if vxsqrt_flag=1 then SetFX(FPSCR.VXSQRT)
   if xx_flag=1     then SetFX(FPSCR.XX)

   ex_flag ← ex_flag | (FPSCR.VE & vxsnan_flag)
                     | (FPSCR.VE & vxsqrt_flag)
                     | (FPSCR.XE & xx_flag
end

if ex_flag=0 then VSR[32×TX+T] ← vresult
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

For each integer value i from 0 to 3, do the following.

Let src be the single-precision floating-point operand in word element i of VSR[XB].

The unbounded-precision square root of src is produced.

See Table 7.46.

The intermediate result is rounded to single-precision using the rounding mode specified by RN.

See Table 7.9, “Scalar Floating-Point Intermediate Result Handling,” on page 617.

The result is placed into word element i of VSR[XT] in single-precision format.

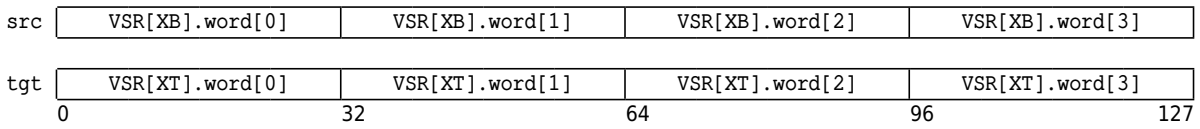
See Table 7.39, “Vector Floating-Point Final Result,” on page 693.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered:

Register	*Field(s)*
FPSCR	FX XX VXSNAN VXSQRT

VSR Data Layout for xvsqrtsp



src							
−Infinity	**−NZF**	**−Zero**	**+Zero**	**+NZF**	**+Infinity**	**QNaN**	**SNaN**
v ← dQNaN vxsqrt_flag ← 1	v ← dQNaN vxsqrt_flag ← 1	v ← +Zero	v ← +Zero	v ← SQRT(src)	v ← +Infinity	v ← src	v ← Q(src) vxsnan_flag ← 1

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 signficand precision and unbounded exponent range.

Table 7.46: Actions for xvsqrtsp

VSX Vector Subtract Double-Precision XX3-form

xvsubdp XT,XA,XB

60	T	A	B	104	AX	BX	TX
0	6	11	16	21	29	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

ex_flag ← 0b0

do i = 0 to 1
   reset_xflags()

   src1 ← bfp_CONVERT_FROM_BFP64(VSR[32×AX+A].dword[i])
   src2 ← bfp_CONVERT_FROM_BFP64(VSR[32×BX+B].dword[i])
   v    ← bfp_ADD(src1,bfp_NEGATE(src2))
   rnd  ← bfp_ROUND_TO_BFP64(0b0,FPSCR.RN,v)

   vresult.dword[i] ← bfp64_CONVERT_FROM_BFP(rnd)

   if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
   if vxisi_flag=1 then SetFX(FPSCR.VXISI)
   if ox_flag=1 then SetFX(FPSCR.OX)
   if ux_flag=1 then SetFX(FPSCR.UX)
   if xx_flag=1 then SetFX(FPSCR.XX)

   ex_flag ← ex_flag | (FPSCR.VE & vxsnan_flag)
                     | (FPSCR.VE & vxisi_flag)
                     | (FPSCR.OE & ox_flag)
                     | (FPSCR.UE & ux_flag)
                     | (FPSCR.XE & xx_flag)
end

if ex_flag=0 then VSR[32×TX+T] ← vresult
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

For each integer value i from 0 to 1, do the following.

Let src1 be the double-precision floating-point operand in doubleword element i of VSR[XA].
Let src2 be the double-precision floating-point operand in doubleword element i of VSR[XB].

src2 is negated and added[77] to src1, producing a sum having unbounded range and precision.

The sum is normalized[78].

See Table 7.47.

The intermediate result is rounded to double-precision using the rounding mode specified by RN.

See Table 7.9, “Scalar Floating-Point Intermediate Result Handling,” on page 617.

The result is placed into doubleword element i of VSR[XT] in double-precision format.

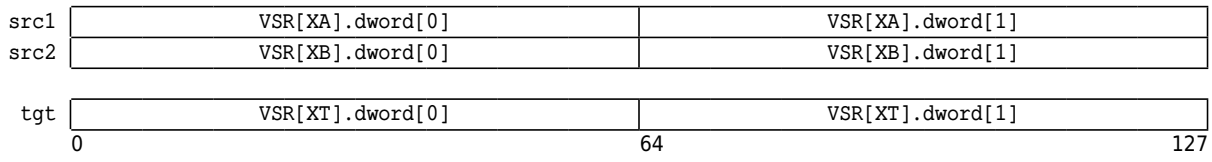
See Table 7.39, “Vector Floating-Point Final Result,” on page 693.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered:

Register	*Field(s)*
FPSCR	FX OX UX XX VXSNAN VXISI

VSR Data Layout for xvsubdp



[77] 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.

[78] 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.

		src2							
		−Infinity	**−NZF**	**−Zero**	**+Zero**	**+NZF**	**+Infinity**	**QNaN**	**SNaN**
src1	**−Infinity**	v ← dQNaN vxisi_flag ← 1	v ← −Infinity	v ← −Infinity	v ← −Infinity	v ← −Infinity	v ← −Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	−NZF	v ← +Infinity	v ← S(src1,src2)	v ← src1	v ← src1	v ← S(src1,src2)	v ← −Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	−Zero	v ← +Infinity	v ← −src2	v ← Rezd	v ← −Zero	v ← −src2	v ← −Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+Zero	v ← +Infinity	v ← −src2	v ← +Zero	v ← Rezd	v ← −src2	v ← −Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+NZF	v ← +Infinity	v ← S(src1,src2)	v ← src1	v ← src1	v ← S(src1,src2)	v ← −Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← dQNaN vxisi_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	QNaN	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1 vxsnan_flag ← 1
	SNaN	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

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 signficand precision and unbounded exponent range.

Table 7.47: Actions for xvsubdp

VSX Vector Subtract Single-Precision XX3-form

xvsubsp XT,XA,XB

60	T	A	B	72	AX	BX	TX
0	6	11	16	21	29	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

ex_flag ← 0b0

do i = 0 to 3
   reset_xflags()

   src1 ← bfp_CONVERT_FROM_BFP32(VSR[32×AX+A].word[i])
   src2 ← bfp_CONVERT_FROM_BFP32(VSR[32×BX+B].word[i])
   v    ← bfp_ADD(src1,bfp_NEGATE(src2))
   rnd  ← bfp_ROUND_TO_BFP32(FPSCR.RN,v)

   vresult.word[i] ← bfp32_CONVERT_FROM_BFP(rnd)

   if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
   if vxisi_flag=1  then SetFX(FPSCR.VXISI)
   if ox_flag=1     then SetFX(FPSCR.OX)
   if ux_flag=1     then SetFX(FPSCR.UX)
   if xx_flag=1     then SetFX(FPSCR.XX)

   ex_flag ← ex_flag | (FPSCR.VE & vxsnan_flag)
                     | (FPSCR.VE & vxisi_flag)
                     | (FPSCR.OE & ox_flag)
                     | (FPSCR.UE & ux_flag)
                     | (FPSCR.XE & xx_flag)
end

if ex_flag=0 then VSR[32×TX+T] ← vresult
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

For each integer value i from 0 to 3, do the following.

Let src1 be the single-precision floating-point operand in word element i of VSR[XA].
Let src2 be the single-precision floating-point operand in word element i of VSR[XB].

src2 is negated and added[79] to src1, producing a sum having unbounded range and precision.

The sum is normalized[80].

See Table 7.48.

The intermediate result is rounded to single-precision using the rounding mode specified by RN.

See Table 7.9, “Scalar Floating-Point Intermediate Result Handling,” on page 617.

The result is placed into word element i of VSR[XT] in single-precision format.

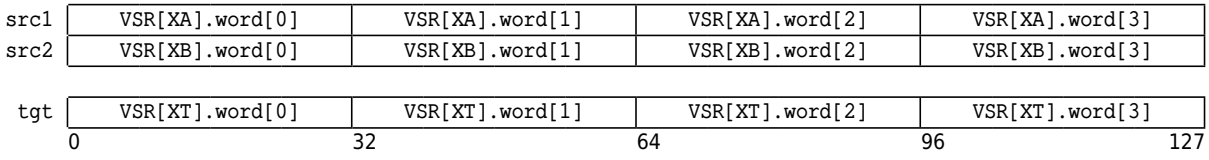
See Table 7.39, “Vector Floating-Point Final Result,” on page 693.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered:

Register	*Field(s)*
FPSCR	FX OX UX XX VXSNAN VXISI

VSR Data Layout for xvsubsp



[79] 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.

[80] 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.

		src2							
		−Infinity	**−NZF**	**−Zero**	**+Zero**	**+NZF**	**+Infinity**	**QNaN**	**SNaN**
src1	**−Infinity**	v ← dQNaN vxisi_flag ← 1	v ← −Infinity	v ← −Infinity	v ← −Infinity	v ← −Infinity	v ← −Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	−NZF	v ← +Infinity	v ← S(src1,src2)	v ← src1	v ← src1	v ← S(src1,src2)	v ← −Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	−Zero	v ← +Infinity	v ← −src2	v ← Rezd	v ← −Zero	v ← −src2	v ← −Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+Zero	v ← +Infinity	v ← −src2	v ← +Zero	v ← Rezd	v ← −src2	v ← −Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+NZF	v ← +Infinity	v ← S(src1,src2)	v ← src1	v ← src1	v ← S(src1,src2)	v ← −Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← dQNaN vxisi_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	QNaN	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1 vxsnan_flag ← 1
	SNaN	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

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 signficand precision and unbounded exponent range.

Table 7.48: Actions for xvsubsp

VSX Vector Multiply-Add Type-A Double-Precision XX3-form

xvmaddadp XT,XA,XB

60	T	A	B	97	AX	BX	TX
0	6	11	16	21	29	30	31

VSX Vector Multiply-Add Type-M Double-Precision XX3-form

xvmaddmdp XT,XA,XB

60	T	A	B	105	AX	BX	TX
0	6	11	16	21	29	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

ex_flag ← 0b0

do i = 0 to 1
   reset_xflags()

  if ‘‘xvmaddadp’’ then do
    src1 ← bfp_CONVERT_FROM_BFP64(VSR[32×AX+A].dword[i])
    src2 ← bfp_CONVERT_FROM_BFP64(VSR[32×TX+T].dword[i])
    src3 ← bfp_CONVERT_FROM_BFP64(VSR[32×BX+B].dword[i])
  end
  else do
    src1 ← bfp_CONVERT_FROM_BFP64(VSR[32×AX+A].dword[i])
    src2 ← bfp_CONVERT_FROM_BFP64(VSR[32×BX+B].dword[i])
    src3 ← bfp_CONVERT_FROM_BFP64(VSR[32×TX+T].dword[i])
  end

  v   ← bfp_MULTIPLY_ADD(src1,src3,src2)
  rnd ← bfp_ROUND_TO_BFP64(0b0,FPSCR.RN,v)

  vresult.dword[i] ← bfp64_CONVERT_FROM_BFP(rnd)

  if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
  if vximz_flag=1 then SetFX(FPSCR.VXIMZ)
  if vxisi_flag=1 then SetFX(FPSCR.VXISI)
  if ox_flag=1 then SetFX(FPSCR.OX)
  if ux_flag=1 then SetFX(FPSCR.UX)
  if xx_flag=1 then SetFX(FPSCR.XX)

  ex_flag ← ex_flag | (FPSCR.VE & vxsnan_flag)
                    | (FPSCR.VE & vximz_flag)
                    | (FPSCR.VE & vxisi_flag)
                    | (FPSCR.OE & ox_flag)
                    | (FPSCR.UE & ux_flag)
                    | (FPSCR.XE & xx_flag)
end

if ex_flag=0 then VSR[32×TX+T] ← result
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

For each integer value i from 0 to 1, do the following.

For ***xvmaddadp***, do the following.

- Let src1 be the double-precision floating-point operand in doubleword element i of VSR[XA].
- Let src2 be the double-precision floating-point operand in doubleword element i of VSR[XT].
- Let src3 be the double-precision floating-point operand in doubleword element i of VSR[XB].

For ***xvmaddmdp***, do the following.

- Let src1 be the double-precision floating-point operand in doubleword element i of VSR[XA].
- Let src2 be the double-precision floating-point operand in doubleword element i of VSR[XB].
- Let src3 be the double-precision floating-point operand in doubleword element i of VSR[XT].

src1 is multiplied[81] by src3, producing a product having unbounded range and precision.

See part 1 of Table 7.49.

src2 is added[82] to the product, producing a sum having unbounded range and precision.

The sum is normalized[83].

See part 2 of Table 7.49.

The intermediate result is rounded to double-precision using the rounding mode specified by RN.

See Table 7.9, “Scalar Floating-Point Intermediate Result Handling,” on page 617.

The result is placed into doubleword element i of VSR[XT] in double-precision format.

See Table 7.39, “Vector Floating-Point Final Result,” on page 693.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered:

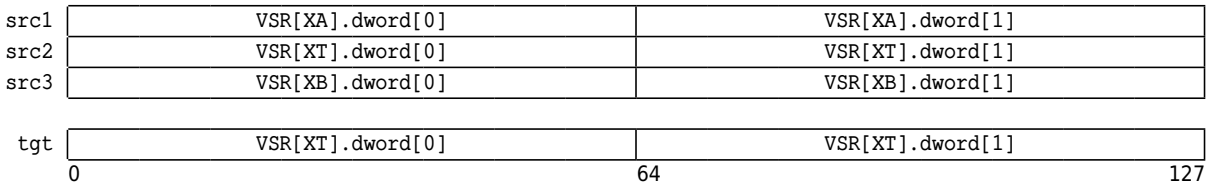
Register	*Field(s)*
FPSCR	FX OX UX XX VXSNAN VXISI VXIMZ

[81] Floating-point multiplication is based on exponent addition and multiplication of the significands.

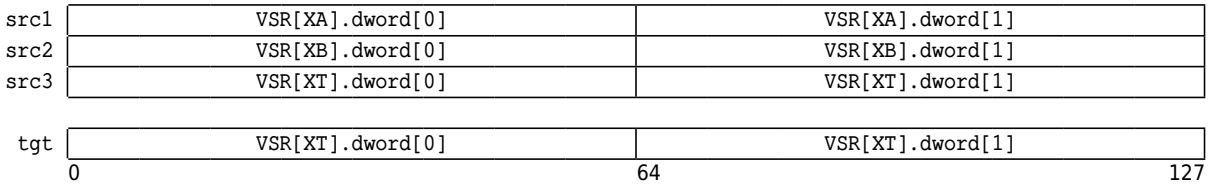
[82] Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are 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.

[83] 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 xvmaddadp



VSR Data Layout for xvmaddmdp



Part 1: Multiply	src1 \ src3	−Infinity	−NZF	−Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
	−Infinity	p ← +Infinity	p ← +Infinity	p ← dQNaN vximz_flag ← 1	p ← dQNaN vximz_flag ← 1	p ← −Infinity	p ← −Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	−NZF	p ← +Infinity	p ← M(src1,src3)	p ← +Zero	p ← −Zero	p ← M(src1,src3)	p ← −Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	−Zero	p ← dQNaN vximz_flag ← 1	p ← +Zero	p ← +Zero	p ← −Zero	p ← −Zero	p ← dQNaN vximz_flag ← 1	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	+Zero	p ← dQNaN vximz_flag ← 1	p ← −Zero	p ← −Zero	p ← +Zero	p ← +Zero	p ← dQNaN vximz_flag ← 1	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	+NZF	p ← −Infinity	p ← M(src1,src3)	p ← −Zero	p ← +Zero	p ← M(src1,src3)	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	+Infinity	p ← −Infinity	p ← −Infinity	p ← dQNaN vximz_flag ← 1	p ← dQNaN vximz_flag ← 1	p ← +Infinity	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	QNaN	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1 vxsnan_flag ← 1
	SNaN	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1

Part 2: Add	p \ src2	−Infinity	−NZF	−Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
	−Infinity	v ← −Infinity	v ← −Infinity	v ← −Infinity	v ← −Infinity	v ← −Infinity	v ← dQNaN vxisi_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	−NZF	v ← −Infinity	v ← A(p,src2)	v ← p	v ← p	v ← A(p,src2)	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	−Zero	v ← −Infinity	v ← src2	v ← −Zero	v ← Rezd	v ← src2	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+Zero	v ← −Infinity	v ← src2	v ← Rezd	v ← +Zero	v ← src2	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+NZF	v ← −Infinity	v ← A(p,src2)	v ← p	v ← p	v ← A(p,src2)	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+Infinity	v ← dQNaN vxisi_flag ← 1	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	QNaN & src1 is a NaN	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p vxsnan_flag ← 1
	QNaN & src1 not a NaN	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← src2	v ← Q(src2) vxsnan_flag ← 1

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}`). For ***xvmaddmdp***, the double-precision floating-point value in doubleword element `i` of `VSR[XB]` (where `i={0,1}`).
src3	For ***xvmaddadp***, the double-precision floating-point value in doubleword element `i` of `VSR[XB]` (where `i={0,1}`). For ***xvmaddmdp***, the double-precision floating-point value in doubleword element `i` of `VSR[XT]` (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 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 7.49: Actions for xvmadd(a|m)dp

VSX Vector Multiply-Add Type-A Single-Precision XX3-form

xvmaddasp XT,XA,XB

60	T	A	B	65	AX	BX	TX
0	6	11	16	21	29	30	31

VSX Vector Multiply-Add Type-M Single-Precision XX3-form

xvmaddmsp XT,XA,XB

60	T	A	B	73	AX	BX	TX
0	6	11	16	21	29	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

ex_flag ← 0b0

do i = 0 to 3
   reset_xflags()

  if ‘‘xvmaddasp’’ then do
    src1 ← bfp_CONVERT_FROM_BFP32(VSR[32×AX+A].word[i])
    src2 ← bfp_CONVERT_FROM_BFP32(VSR[32×TX+T].word[i])
    src3 ← bfp_CONVERT_FROM_BFP32(VSR[32×BX+B].word[i])
  end
  else do
    src1 ← bfp_CONVERT_FROM_BFP32(VSR[32×AX+A].word[i])
    src2 ← bfp_CONVERT_FROM_BFP32(VSR[32×BX+B].word[i])
    src3 ← bfp_CONVERT_FROM_BFP32(VSR[32×TX+T].word[i])
  end

  v   ← bfp_MULTIPLY_ADD(src1,src3,src2)
  rnd ← bfp_ROUND_TO_BFP32(FPSCR.RN,v)

  vresult.word[i] ← bfp32_CONVERT_FROM_BFP(rnd)

  if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
  if vximz_flag=1 then SetFX(FPSCR.VXIMZ)
  if vxisi_flag=1 then SetFX(FPSCR.VXISI)
  if ox_flag=1 then SetFX(FPSCR.OX)
  if ux_flag=1 then SetFX(FPSCR.UX)
  if xx_flag=1 then SetFX(FPSCR.XX)

  ex_flag ← ex_flag | (FPSCR.VE & vxsnan_flag)
                    | (FPSCR.VE & vximz_flag)
                    | (FPSCR.VE & vxisi_flag)
                    | (FPSCR.OE & ox_flag)
                    | (FPSCR.UE & ux_flag)
                    | (FPSCR.XE & xx_flag)
end

if ex_flag=0 then VSR[32×TX+T] ← result
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

For each integer value i from 0 to 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[XT].
- Let src3 be the single-precision floating-point operand in word element i of VSR[XB].

For ***xvmaddmsp***, do the following.

- Let src1 be the single-precision floating-point operand in word element i of VSR[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[84] by src3, producing a product having unbounded range and precision.

See part 1 of Table 7.50.

src2 is added[85] to the product, producing a sum having unbounded range and precision.

The sum is normalized[86].

See part 2 of Table 7.50.

The intermediate result is rounded to single-precision using the rounding mode specified by RN.

See Table 7.9, “Scalar Floating-Point Intermediate Result Handling,” on page 617.

The result is placed into word element i of VSR[XT] in single-precision format.

See Table 7.39, “Vector Floating-Point Final Result,” on page 693.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered:

Register	*Field(s)*
FPSCR	FX OX UX XX VXSNAN VXISI VXIMZ

[84] Floating-point multiplication is based on exponent addition and multiplication of the significands.

[85] 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.

[86] Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.

VSR Data Layout for xvmaddasp

src1	VSR[XA].word[0]	VSR[XA].word[1]	VSR[XA].word[2]	VSR[XA].word[3]
src2	VSR[XT].word[0]	VSR[XT].word[1]	VSR[XT].word[2]	VSR[XT].word[3]
src3	VSR[XB].word[0]	VSR[XB].word[1]	VSR[XB].word[2]	VSR[XB].word[3]
tgt	VSR[XT].word[0]	VSR[XT].word[1]	VSR[XT].word[2]	VSR[XT].word[3]

0 32 64 96 127

VSR Data Layout for xvmaddmsp

src1	VSR[XA].word[0]	VSR[XA].word[1]	VSR[XA].word[2]	VSR[XA].word[3]
src2	VSR[XB].word[0]	VSR[XB].word[1]	VSR[XB].word[2]	VSR[XB].word[3]
src3	VSR[XT].word[0]	VSR[XT].word[1]	VSR[XT].word[2]	VSR[XT].word[3]
tgt	VSR[XT].word[0]	VSR[XT].word[1]	VSR[XT].word[2]	VSR[XT].word[3]

0 32 64 96 127

Part 1: Multiply

src1 \ src3	−Infinity	−NZF	−Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
−Infinity	p ← +Infinity	p ← +Infinity	p ← dQNaN vximz_flag ← 1	p ← dQNaN vximz_flag ← 1	p ← −Infinity	p ← −Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
−NZF	p ← +Infinity	p ← M(src1,src3)	p ← +Zero	p ← −Zero	p ← M(src1,src3)	p ← −Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
−Zero	p ← dQNaN vximz_flag ← 1	p ← +Zero	p ← +Zero	p ← −Zero	p ← −Zero	p ← dQNaN vximz_flag ← 1	p ← src3	p ← Q(src3) vxsnan_flag ← 1
+Zero	p ← dQNaN vximz_flag ← 1	p ← −Zero	p ← −Zero	p ← +Zero	p ← +Zero	p ← dQNaN vximz_flag ← 1	p ← src3	p ← Q(src3) vxsnan_flag ← 1
+NZF	p ← −Infinity	p ← M(src1,src3)	p ← −Zero	p ← +Zero	p ← M(src1,src3)	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
+Infinity	p ← −Infinity	p ← −Infinity	p ← dQNaN vximz_flag ← 1	p ← dQNaN vximz_flag ← 1	p ← +Infinity	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
QNaN	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1 vxsnan_flag ← 1
SNaN	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1

Part 2: Add

p \ src2	−Infinity	−NZF	−Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
−Infinity	v ← −Infinity	v ← −Infinity	v ← −Infinity	v ← −Infinity	v ← −Infinity	v ← dQNaN vxisi_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
−NZF	v ← −Infinity	v ← A(p,src2)	v ← p	v ← p	v ← A(p,src2)	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
−Zero	v ← −Infinity	v ← src2	v ← −Zero	v ← Rezd	v ← src2	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Zero	v ← −Infinity	v ← src2	v ← Rezd	v ← +Zero	v ← src2	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+NZF	v ← −Infinity	v ← A(p,src2)	v ← p	v ← p	v ← A(p,src2)	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Infinity	v ← dQNaN vxisi_flag ← 1	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
QNaN & src1 is a NaN	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p vxsnan_flag ← 1
QNaN & src1 not a NaN	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← src2	v ← Q(src2) vxsnan_flag ← 1

Explanation:

src1	The single-precision floating-point value in word element `i` of `VSR[XA]` (where `i={0,1,2,3}`).
src2	For ***xvmaddasp***, the single-precision floating-point value in word element `i` of `VSR[XT]` (where `i={0,1,2,3}`). For ***xvmaddmsp***, the single-precision floating-point value in word element `i` of `VSR[XB]` (where `i={0,1,2,3}`).
src3	For ***xvmaddasp***, the single-precision floating-point value in word element `i` of `VSR[XB]` (where `i={0,1,2,3}`). For ***xvmaddmsp***, the single-precision floating-point value in word element `i` of `VSR[XT]` (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.

Table 7.50: Actions for xvmadd(a|m)sp

VSX Vector Multiply-Subtract Type-A Double-Precision XX3-form

xvmsubadp XT,XA,XB

60	T	A	B	113	AX	BX	TX
0	6	11	16	21	29	30	31

VSX Vector Multiply-Subtract Type-M Double-Precision XX3-form

xvmsubmdp XT,XA,XB

60	T	A	B	121	AX	BX	TX
0	6	11	16	21	29	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

ex_flag ← 0b0

do i = 0 to 1
   reset_xflags()

  if ‘‘xvmsubadp’’ then do
    src1 ← bfp_CONVERT_FROM_BFP64(VSR[32×AX+A].dword[i])
    src2 ← bfp_CONVERT_FROM_BFP64(VSR[32×TX+T].dword[i])
    src3 ← bfp_CONVERT_FROM_BFP64(VSR[32×BX+B].dword[i])
  end
  else do
    src1 ← bfp_CONVERT_FROM_BFP64(VSR[32×AX+A].dword[i])
    src2 ← bfp_CONVERT_FROM_BFP64(VSR[32×BX+B].dword[i])
    src3 ← bfp_CONVERT_FROM_BFP64(VSR[32×TX+T].dword[i])
  end

  v   ← bfp_MULTIPLY_ADD(src1,src3,bfp_NEGATE(src2))
  rnd ← bfp_ROUND_TO_BFP64(0b0,FPSCR.RN,v)

  vresult.dword[i] ← bfp64_CONVERT_FROM_BFP(rnd)

  if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
  if vximz_flag=1 then SetFX(FPSCR.VXIMZ)
  if vxisi_flag=1 then SetFX(FPSCR.VXISI)
  if ox_flag=1 then SetFX(FPSCR.OX)
  if ux_flag=1 then SetFX(FPSCR.UX)
  if xx_flag=1 then SetFX(FPSCR.XX)

  ex_flag ← ex_flag | (FPSCR.VE & vxsnan_flag)
                    | (FPSCR.VE & vximz_flag)
                    | (FPSCR.VE & vxisi_flag)
                    | (FPSCR.OE & ox_flag)
                    | (FPSCR.UE & ux_flag)
                    | (FPSCR.XE & xx_flag)
end

if ex_flag=0 then VSR[32×TX+T] ← result
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

For each integer value i from 0 to 1, do the following.

For ***xvmsubadp***, do the following.

- Let src1 be the double-precision floating-point operand in doubleword element i of VSR[XA].
- Let src2 be the double-precision floating-point operand in doubleword element i of VSR[XT].
- Let src3 be the double-precision floating-point operand in doubleword element i of VSR[XB].

For ***xvmsubmdp***, do the following.

- Let src1 be the double-precision floating-point operand in doubleword element i of VSR[XA].
- Let src2 be the double-precision floating-point operand in doubleword element i of VSR[XB].
- Let src3 be the double-precision floating-point operand in doubleword element i of VSR[XT].

src1 is multiplied[87] by src3, producing a product having unbounded range and precision.

See part 1 of Table 7.51.

src2 is negated and added[88] to the product, producing a sum having unbounded range and precision.

The sum is normalized[89].

See part 2 of Table 7.51.

The intermediate result is rounded to double-precision using the rounding mode specified by RN.

See Table 7.9, “Scalar Floating-Point Intermediate Result Handling,” on page 617.

The result is placed into doubleword element i of VSR[XT] in double-precision format.

See Table 7.39, “Vector Floating-Point Final Result,” on page 693.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered:

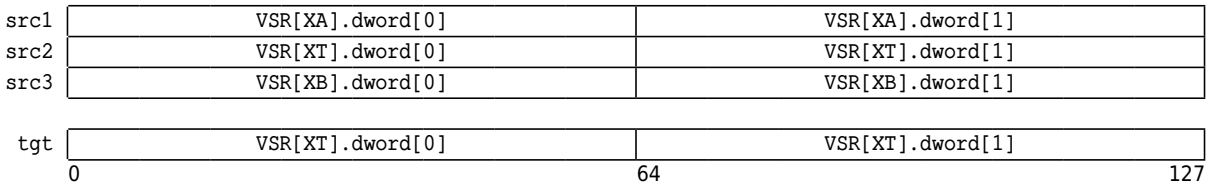
Register	*Field(s)*
FPSCR	FX OX UX XX VXSNAN VXISI VXIMZ

[87] Floating-point multiplication is based on exponent addition and multiplication of the significands.

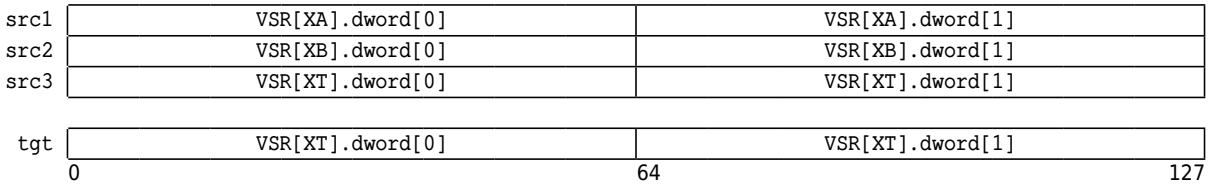
[88] Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are 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.

[89] Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.

VSR Data Layout for xvmsubadp



VSR Data Layout for xvmsubmdp



Part 1: Multiply	src3							
src1	−Infinity	−NZF	−Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
−Infinity	p ← +Infinity	p ← +Infinity	p ← dQNaN vximz_flag ← 1	p ← dQNaN vximz_flag ← 1	p ← −Infinity	p ← −Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
−NZF	p ← +Infinity	p ← M(src1,src3)	p ← +Zero	p ← −Zero	p ← M(src1,src3)	p ← −Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
−Zero	p ← dQNaN vximz_flag ← 1	p ← +Zero	p ← +Zero	p ← −Zero	p ← −Zero	p ← dQNaN vximz_flag ← 1	p ← src3	p ← Q(src3) vxsnan_flag ← 1
+Zero	p ← dQNaN vximz_flag ← 1	p ← −Zero	p ← −Zero	p ← +Zero	p ← +Zero	p ← dQNaN vximz_flag ← 1	p ← src3	p ← Q(src3) vxsnan_flag ← 1
+NZF	p ← −Infinity	p ← M(src1,src3)	p ← −Zero	p ← +Zero	p ← M(src1,src3)	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
+Infinity	p ← −Infinity	p ← −Infinity	p ← dQNaN vximz_flag ← 1	p ← dQNaN vximz_flag ← 1	p ← +Infinity	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
QNaN	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1 vxsnan_flag ← 1
SNaN	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1

Part 2: Subtract	src2							
p	−Infinity	−NZF	−Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
−Infinity	v ← dQNaN vxisi_flag ← 1	v ← −Infinity	v ← −Infinity	v ← −Infinity	v ← −Infinity	v ← −Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
−NZF	v ← +Infinity	v ← S(p,src2)	v ← p	v ← p	v ← S(p,src2)	v ← −Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
−Zero	v ← +Infinity	v ← −src2	v ← Rezd	v ← −Zero	v ← −src2	v ← −Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Zero	v ← +Infinity	v ← −src2	v ← +Zero	v ← Rezd	v ← −src2	v ← −Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+NZF	v ← +Infinity	v ← S(p,src2)	v ← p	v ← p	v ← S(p,src2)	v ← −Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← dQNaN vxisi_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
QNaN & src1 is a NaN	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p vxsnan_flag ← 1
QNaN & src1 not a NaN	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← src2	v ← Q(src2) vxsnan_flag ← 1

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}`). For ***xvmsubmdp***, the double-precision floating-point value in doubleword element `i` of `VSR[XB]` (where `i={0,1}`).
src3	For ***xvmsubadp***, the double-precision floating-point value in doubleword element `i` of `VSR[XB]` (where `i={0,1}`). For ***xvmsubmdp***, the double-precision floating-point value in doubleword element `i` of `VSR[XT]` (where `i={0,1}`).
dQNaN	Default quiet NaN (`0x7FF8_0000_0000_0000`).
NZF	Nonzero finite number.
Rezd	Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs). Can also occur with two nonzero finite number source operands.
Q(x)	Return a QNaN with the payload of x.
S(x,y)	Return the normalized sum of floating-point value x and negated floating-point value y, having unbounded range and precision. Note: If x = y, v is considered to be an exact-zero-difference result (Rezd).
M(x,y)	Return the normalized product of floating-point value x and floating-point value y, having unbounded range and precision.
p	The intermediate product having unbounded range and precision.
v	The intermediate result having unbounded range and precision.

Table 7.51: Actions for xvmsub(a|m)dp

VSX Vector Multiply-Subtract Type-A Single-Precision XX3-form

xvmsubasp XT,XA,XB

60	T	A	B	81	AX	BX	TX
0	6	11	16	21	29	30	31

VSX Vector Multiply-Subtract Type-M Single-Precision XX3-form

xvmsubmsp XT,XA,XB

60	T	A	B	89	AX	BX	TX
0	6	11	16	21	29	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

ex_flag ← 0b0

do i = 0 to 3
   reset_xflags()

  if ‘‘xvmsubasp’’ then do
    src1 ← bfp_CONVERT_FROM_BFP32(VSR[32×AX+A].word[i])
    src2 ← bfp_CONVERT_FROM_BFP32(VSR[32×TX+T].word[i])
    src3 ← bfp_CONVERT_FROM_BFP32(VSR[32×BX+B].word[i])
  end
  else do
    src1 ← bfp_CONVERT_FROM_BFP32(VSR[32×AX+A].word[i])
    src2 ← bfp_CONVERT_FROM_BFP32(VSR[32×BX+B].word[i])
    src3 ← bfp_CONVERT_FROM_BFP32(VSR[32×TX+T].word[i])
  end

  v   ← bfp_MULTIPLY_ADD(src1,src3,bfp_NEGATE(src2))
  rnd ← bfp_ROUND_TO_BFP32(FPSCR.RN,v)

  vresult.word[i] ← bfp32_CONVERT_FROM_BFP(rnd)

  if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
  if vximz_flag=1 then SetFX(FPSCR.VXIMZ)
  if vxisi_flag=1 then SetFX(FPSCR.VXISI)
  if ox_flag=1 then SetFX(FPSCR.OX)
  if ux_flag=1 then SetFX(FPSCR.UX)
  if xx_flag=1 then SetFX(FPSCR.XX)

  ex_flag ← ex_flag | (FPSCR.VE & vxsnan_flag)
                    | (FPSCR.VE & vximz_flag)
                    | (FPSCR.VE & vxisi_flag)
                    | (FPSCR.OE & ox_flag)
                    | (FPSCR.UE & ux_flag)
                    | (FPSCR.XE & xx_flag)
end

if ex_flag=0 then VSR[32×TX+T] ← result
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

For each integer value i from 0 to 3, do the following.

For ***xvmsubasp***, do the following.

- Let src1 be the single-precision floating-point operand in word element i of VSR[XA].
- Let src2 be the single-precision floating-point operand in word element i of VSR[XT].
- Let src3 be the single-precision floating-point operand in word element i of VSR[XB].

For ***xvmsubmsp***, 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[90] by src3, producing a product having unbounded range and precision.

See part 1 of Table 7.52.

src2 is negated and added[91] to the product, producing a sum having unbounded range and precision.

The sum is normalized[92].

See part 2 of Table 7.52.

The intermediate result is rounded to single-precision using the rounding mode specified by RN.

See Table 7.9, “Scalar Floating-Point Intermediate Result Handling,” on page 617.

The result is placed into word element i of VSR[XT] in single-precision format.

See Table 7.39, “Vector Floating-Point Final Result,” on page 693.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered:

Register	*Field(s)*
FPSCR	FX OX UX XX VXSNAN VXISI VXIMZ

[90] Floating-point multiplication is based on exponent addition and multiplication of the significands.

[91] 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.

[92] Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.

VSR Data Layout for xvmsubasp

src1	VSR[XA].word[0]	VSR[XA].word[1]	VSR[XA].word[2]	VSR[XA].word[3]
src2	VSR[XT].word[0]	VSR[XT].word[1]	VSR[XT].word[2]	VSR[XT].word[3]
src3	VSR[XB].word[0]	VSR[XB].word[1]	VSR[XB].word[2]	VSR[XB].word[3]

tgt	VSR[XT].word[0]	VSR[XT].word[1]	VSR[XT].word[2]	VSR[XT].word[3]
	0	32	64	96 127

VSR Data Layout for xvmsubmsp

src1	VSR[XA].word[0]	VSR[XA].word[1]	VSR[XA].word[2]	VSR[XA].word[3]
src2	VSR[XB].word[0]	VSR[XB].word[1]	VSR[XB].word[2]	VSR[XB].word[3]
src3	VSR[XT].word[0]	VSR[XT].word[1]	VSR[XT].word[2]	VSR[XT].word[3]

tgt	VSR[XT].word[0]	VSR[XT].word[1]	VSR[XT].word[2]	VSR[XT].word[3]
	0	32	64	96 127

Part 1: Multiply		src3							
		−Infinity	−NZF	−Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
src1	−Infinity	p ← +Infinity	p ← +Infinity	p ← dQNaN vximz_flag ← 1	p ← dQNaN vximz_flag ← 1	p ← −Infinity	p ← −Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	−NZF	p ← +Infinity	p ← M(src1,src3)	p ← +Zero	p ← −Zero	p ← M(src1,src3)	p ← −Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	−Zero	p ← dQNaN vximz_flag ← 1	p ← +Zero	p ← +Zero	p ← −Zero	p ← −Zero	p ← dQNaN vximz_flag ← 1	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	+Zero	p ← dQNaN vximz_flag ← 1	p ← −Zero	p ← −Zero	p ← +Zero	p ← +Zero	p ← dQNaN vximz_flag ← 1	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	+NZF	p ← −Infinity	p ← M(src1,src3)	p ← −Zero	p ← +Zero	p ← M(src1,src3)	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	+Infinity	p ← −Infinity	p ← −Infinity	p ← dQNaN vximz_flag ← 1	p ← dQNaN vximz_flag ← 1	p ← +Infinity	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	QNaN	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1 vxsnan_flag ← 1
	SNaN	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1

Part 2: Subtract		src2							
		−Infinity	−NZF	−Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
p	−Infinity	v ← dQNaN vxisi_flag ← 1	v ← −Infinity	v ← −Infinity	v ← −Infinity	v ← −Infinity	v ← −Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	−NZF	v ← +Infinity	v ← S(p,src2)	v ← p	v ← p	v ← S(p,src2)	v ← −Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	−Zero	v ← +Infinity	v ← −src2	v ← Rezd	v ← −Zero	v ← −src2	v ← −Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+Zero	v ← +Infinity	v ← −src2	v ← +Zero	v ← Rezd	v ← −src2	v ← −Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+NZF	v ← +Infinity	v ← S(p,src2)	v ← p	v ← p	v ← S(p,src2)	v ← −Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← dQNaN vxisi_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	QNaN & src1 is a NaN	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p vxsnan_flag ← 1
	QNaN & src1 not a NaN	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← src2	v ← Q(src2) vxsnan_flag ← 1

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}`). For ***xvmsubmsp***, the single-precision floating-point value in word element `i` of `VSR[XB]` (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}`). For ***xvmsubmsp***, the single-precision floating-point value in word element `i` of `VSR[XT]` (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. 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 7.52: Actions for xvmsub(a|m)sp

VSX Vector Negative Multiply-Add Type-A Double-Precision XX3-form

xvnmaddadp XT,XA,XB

60	T	A	B	225	AX	BX	TX
0	6	11	16	21	29	30	31

VSX Vector Negative Multiply-Add Type-M Double-Precision XX3-form

xvnmaddmdp XT,XA,XB

60	T	A	B	233	AX	BX	TX
0	6	11	16	21	29	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

ex_flag ← 0b0

do i = 0 to 1
  reset_xflags()

  if ‘‘xvnmaddadp’’ then do
    src1 ← bfp_CONVERT_FROM_BFP64(VSR[32×AX+A].dword[i])
    src2 ← bfp_CONVERT_FROM_BFP64(VSR[32×TX+T].dword[i])
    src3 ← bfp_CONVERT_FROM_BFP64(VSR[32×BX+B].dword[i])
  end
  else do
    src1 ← bfp_CONVERT_FROM_BFP64(VSR[32×AX+A].dword[i])
    src2 ← bfp_CONVERT_FROM_BFP64(VSR[32×BX+B].dword[i])
    src3 ← bfp_CONVERT_FROM_BFP64(VSR[32×TX+T].dword[i])
  end

  v   ← bfp_MULTIPLY_ADD(src1,src3,src2)
  rnd ← bfp_NEGATE(bfp_ROUND_TO_BFP64(FPSCR.RN,v))

  vresult.dword[i] ← bfp64_CONVERT_FROM_BFP(rnd)

  if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
  if vximz_flag=1 then SetFX(FPSCR.VXIMZ)
  if vxisi_flag=1 then SetFX(FPSCR.VXISI)
  if ox_flag=1 then SetFX(FPSCR.OX)
  if ux_flag=1 then SetFX(FPSCR.UX)
  if xx_flag=1 then SetFX(FPSCR.XX)

  ex_flag ← ex_flag | (FPSCR.VE & vxsnan_flag)
                    | (FPSCR.VE & vximz_flag)
                    | (FPSCR.VE & vxisi_flag)
                    | (FPSCR.OE & ox_flag)
                    | (FPSCR.UE & ux_flag)
                    | (FPSCR.XE & xx_flag)
end

if ex_flag=0 then VSR[32×TX+T] ← vresult
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

For each integer value i from 0 to 1, do the following.

For ***xvnmaddadp***, do the following.

- Let src1 be the double-precision floating-point operand in doubleword element i of VSR[XA].
- Let src2 be the double-precision floating-point operand in doubleword element i of VSR[XT].
- Let src3 be the double-precision floating-point operand in doubleword element i of VSR[XB].

For ***xvnmaddmdp***, do the following.

- Let src1 be the double-precision floating-point operand in doubleword element i of VSR[XA].
- Let src2 be the double-precision floating-point operand in doubleword element i of VSR[XB].
- Let src3 be the double-precision floating-point operand in doubleword element i of VSR[XT].

src1 is multiplied[93] by src3, producing a product having unbounded range and precision.

See part 1 of Table 7.53.

src2 is added[94] to the product, producing a sum having unbounded range and precision.

The sum is normalized[95].

See part 2 of Table 7.53.

The intermediate result is rounded to double-precision using the rounding mode specified by RN.

See Table 7.9, “Scalar Floating-Point Intermediate Result Handling,” on page 617.

The result is negated and placed into doubleword element i of VSR[XT] in double-precision format.

See Table 7.54, “Vector Floating-Point Final Result with Negation,” on page 725.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered:

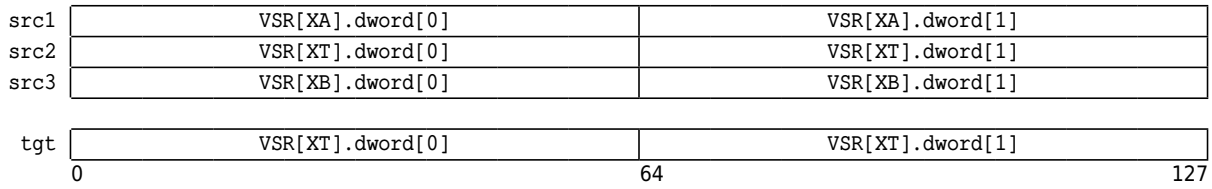
Register	*Field(s)*
FPSCR	FX OX UX XX VXSNAN VXISI VXIMZ

[93] Floating-point multiplication is based on exponent addition and multiplication of the significands.

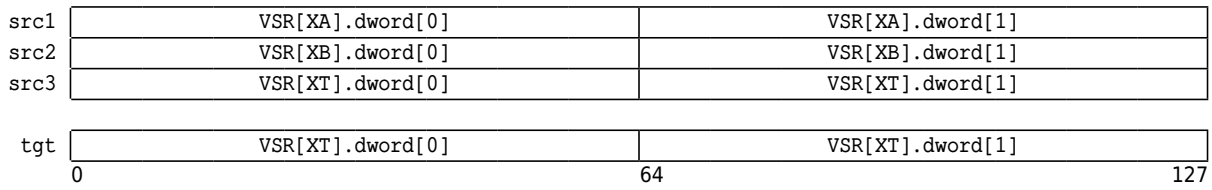
[94] Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are 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.

[95] Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.

VSR Data Layout for xvnmaddadp



VSR Data Layout for xvnmaddmdp



Part 1: Multiply	src3 −Infinity	src3 −NZF	src3 −Zero	src3 +Zero	src3 +NZF	src3 +Infinity	src3 QNaN	src3 SNaN
src1 −Infinity	p ← +Infinity	p ← +Infinity	p ← dQNaN vximz_flag ← 1	p ← dQNaN vximz_flag ← 1	p ← −Infinity	p ← −Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
src1 −NZF	p ← +Infinity	p ← M(src1,src3)	p ← +Zero	p ← −Zero	p ← M(src1,src3)	p ← −Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
src1 −Zero	p ← dQNaN vximz_flag ← 1	p ← +Zero	p ← +Zero	p ← −Zero	p ← −Zero	p ← dQNaN vximz_flag ← 1	p ← src3	p ← Q(src3) vxsnan_flag ← 1
src1 +Zero	p ← dQNaN vximz_flag ← 1	p ← −Zero	p ← −Zero	p ← +Zero	p ← +Zero	p ← dQNaN vximz_flag ← 1	p ← src3	p ← Q(src3) vxsnan_flag ← 1
src1 +NZF	p ← −Infinity	p ← M(src1,src3)	p ← −Zero	p ← +Zero	p ← M(src1,src3)	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
src1 +Infinity	p ← −Infinity	p ← −Infinity	p ← dQNaN vximz_flag ← 1	p ← dQNaN vximz_flag ← 1	p ← +Infinity	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
src1 QNaN	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1 vxsnan_flag ← 1
src1 SNaN	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1

Part 2: Add	src2 −Infinity	src2 −NZF	src2 −Zero	src2 +Zero	src2 +NZF	src2 +Infinity	src2 QNaN	src2 SNaN
p −Infinity	v ← −Infinity	v ← −Infinity	v ← −Infinity	v ← −Infinity	v ← −Infinity	v ← dQNaN vxisi_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
p −NZF	v ← −Infinity	v ← A(p,src2)	v ← p	v ← p	v ← A(p,src2)	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
p −Zero	v ← −Infinity	v ← src2	v ← −Zero	v ← Rezd	v ← src2	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
p +Zero	v ← −Infinity	v ← src2	v ← Rezd	v ← +Zero	v ← src2	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
p +NZF	v ← −Infinity	v ← A(p,src2)	v ← p	v ← p	v ← A(p,src2)	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
p +Infinity	v ← dQNaN vxisi_flag ← 1	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
p QNaN & src1 is a NaN	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p vxsnan_flag ← 1
p QNaN & src1 not a NaN	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← src2	v ← Q(src2) vxsnan_flag ← 1

Explanation:

src1	The double-precision floating-point value in doubleword element `i` of `VSR[XA]` (where `i={0,1}`).
src2	For ***xvnmaddadp***, the double-precision floating-point value in doubleword element `i` of `VSR[XT]` (where `i={0,1}`). For ***xvnmaddmdp***, the double-precision floating-point value in doubleword element `i` of `VSR[XB]` (where `i={0,1}`).
src3	For ***xvnmaddadp***, the double-precision floating-point value in doubleword element `i` of `VSR[XB]` (where `i={0,1}`). For ***xvnmaddmdp***, the double-precision floating-point value in doubleword element `i` of `VSR[XT]` (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 7.53: Actions for xvnmadd(a|m)dp

Case	VE	OE	UE	ZE	XE	vxsnan_flag	vximz_flag	vxisi_flag	Is r inexact? (r ≠ v)	Is r incremented? (|r| > |v|)	Is q inexact? (q ≠ v)	Is q incremented? (|q| > |v|)	Returned Results and Status Setting
Special	–	–	–	–	–	0	0	0	–	–	–	–	T(N(r))
	0	–	–	–	–	–	–	1	–	–	–	–	T(r), fx(VXISI)
	0	–	–	–	–	0	1	–	–	–	–	–	T(r), fx(VXIMZ)
	0	–	–	–	–	1	0	–	–	–	–	–	T(r), fx(VXSNAN)
	0	–	–	–	–	1	1	–	–	–	–	–	T(r), fx(VXSNAN), fx(VXIMZ)
	1	–	–	–	–	–	–	1	–	–	–	–	fx(VXISI), error()
	1	–	–	–	–	0	1	–	–	–	–	–	fx(VXIMZ), error()
	1	–	–	–	–	1	0	–	–	–	–	–	fx(VXSNAN), error()
	1	–	–	–	–	1	1	–	–	–	–	–	fx(VXSNAN), fx(VXIMZ), error()
Normal	–	–	–	–	–	–	–	–	no	–	–	–	T(N(r))
	–	–	–	–	0	–	–	–	yes	no	–	–	T(N(r)), fx(XX)
	–	–	–	–	0	–	–	–	yes	yes	–	–	T(N(r)), fx(XX)
	–	–	–	–	1	–	–	–	yes	no	–	–	T(N(r)), fx(XX), error()
	–	–	–	–	1	–	–	–	yes	yes	–	–	T(N(r)), fx(XX), error()
Overflow	–	0	–	–	0	–	–	–	–	–	–	–	T(N(r)), fx(OX), fx(XX)
	–	0	–	–	1	–	–	–	–	–	–	–	T(N(r)), fx(OX), fx(XX), error()
	–	1	–	–	–	–	–	–	–	–	no	–	fx(OX), error()
	–	1	–	–	–	–	–	–	–	–	yes	no	fx(OX), fx(XX), error()
	–	1	–	–	–	–	–	–	–	–	yes	yes	fx(OX), fx(XX), error()
Tiny	–	–	0	–	–	–	–	–	no	–	–	–	T(N(r))
	–	–	0	–	0	–	–	–	yes	no	–	–	T(N(r)), fx(UX), fx(XX)
	–	–	0	–	0	–	–	–	yes	yes	–	–	T(N(r)), fx(UX), fx(XX)
	–	–	0	–	1	–	–	–	yes	no	–	–	T(N(r)), fx(UX), fx(XX), error()
	–	–	0	–	1	–	–	–	yes	yes	–	–	T(N(r)), fx(UX), fx(XX), error()
	–	–	1	–	–	–	–	–	yes	–	no	–	fx(UX), error()
	–	–	1	–	–	–	–	–	yes	–	yes	no	fx(UX), fx(XX), error()
	–	–	1	–	–	–	–	–	yes	–	yes	yes	fx(UX), fx(XX), error()

Explanation:

–	The results do not depend on this condition.
error()	The system error handler is invoked for the trap-enabled exception if MSR.FE0 and MSR.FE1 are set to any mode other than the ignore-exception mode. Update of the target VSR is suppressed for all vector elements.
fx(x)	FPSCR.FX is set to 1 if FPSCR.x=0. FPSCR.x is set to 1.
q	The value defined in Table 7.9, “Scalar Floating-Point Intermediate Result Handling,” on page 617, signficand rounded to the target precision, unbounded exponent range.
r	The value defined in Table 7.9, “Scalar Floating-Point Intermediate Result Handling,” on page 617, signficand rounded to the target precision, bounded exponent range.
v	The precise intermediate result defined in the instruction having unbounded signficand precision, unbounded exponent range.
N(x)	The value x is is negated by complementing the sign bit of x.
T(x)	The value x is placed in element i of VSR[XT] in the target precision format (where i∈{0,1} for results with 64-bit elements, and i∈{0,1,2,3} for results with 32-bit elements).

Table 7.54: Vector Floating-Point Final Result with Negation

VSX Vector Negative Multiply-Add Type-A Single-Precision XX3-form

xvnmaddasp XT,XA,XB

60	T	A	B	193	AX	BX	TX
0	6	11	16	21	29	30	31

VSX Vector Negative Multiply-Add Type-M Single-Precision XX3-form

xvnmaddmsp XT,XA,XB

60	T	A	B	201	AX	BX	TX
0	6	11	16	21	29	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

ex_flag ← 0b0

do i = 0 to 3
  reset_xflags()

  if ''xvnmaddasp'' then do
    src1 ← bfp_CONVERT_FROM_BFP32(VSR[32×AX+A].word[i])
    src2 ← bfp_CONVERT_FROM_BFP32(VSR[32×TX+T].word[i])
    src3 ← bfp_CONVERT_FROM_BFP32(VSR[32×BX+B].word[i])
  end
  else do
    src1 ← bfp_CONVERT_FROM_BFP32(VSR[32×AX+A].word[i])
    src2 ← bfp_CONVERT_FROM_BFP32(VSR[32×BX+B].word[i])
    src3 ← bfp_CONVERT_FROM_BFP32(VSR[32×TX+T].word[i])
  end

  v   ← bfp_MULTIPLY_ADD(src1,src3,src2)
  rnd ← bfp_NEGATE(bfp_ROUND_TO_BFP32(FPSCR.RN,v))

  vresult.word[i] ← bfp32_CONVERT_FROM_BFP(rnd)

  if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
  if vximz_flag=1 then SetFX(FPSCR.VXIMZ)
  if vxisi_flag=1 then SetFX(FPSCR.VXISI)
  if ox_flag=1 then SetFX(FPSCR.OX)
  if ux_flag=1 then SetFX(FPSCR.UX)
  if xx_flag=1 then SetFX(FPSCR.XX)

  ex_flag ← ex_flag | (FPSCR.VE & vxsnan_flag)
                    | (FPSCR.VE & vximz_flag)
                    | (FPSCR.VE & vxisi_flag)
                    | (FPSCR.OE & ox_flag)
                    | (FPSCR.UE & ux_flag)
                    | (FPSCR.XE & xx_flag)
end

if ex_flag=0 then VSR[32×TX+T] ← vresult
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

For each integer value i from 0 to 3, do the following.

For ***xvnmaddasp***, do the following.

- Let src1 be the single-precision floating-point operand in word element i of VSR[XA].
- Let src2 be the single-precision floating-point operand in word element i of VSR[XT].
- Let src3 be the single-precision floating-point operand in word element i of VSR[XB].

For ***xvnmaddmsp***, 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[96] by src3, producing a product having unbounded range and precision.

See part 1 of Table 7.55.

src2 is added[97] to the product, producing a sum having unbounded range and precision.

The sum is normalized[98].

See part 2 of Table 7.55.

The intermediate result is rounded to single-precision using the rounding mode specified by RN.

See Table 7.9, "Scalar Floating-Point Intermediate Result Handling," on page 617.

The result is negated and placed into word element i of VSR[XT] in single-precision format.

See Table 7.54, "Vector Floating-Point Final Result with Negation," on page 725.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered:

Register	*Field(s)*
FPSCR	FX OX UX XX VXSNAN VXISI VXIMZ

[96] Floating-point multiplication is based on exponent addition and multiplication of the significands.

[97] 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.

[98] Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.

VSR Data Layout for xvnmaddasp

src1	VSR[XA].word[0]	VSR[XA].word[1]	VSR[XA].word[2]	VSR[XA].word[3]
src2	VSR[XT].word[0]	VSR[XT].word[1]	VSR[XT].word[2]	VSR[XT].word[3]
src3	VSR[XB].word[0]	VSR[XB].word[1]	VSR[XB].word[2]	VSR[XB].word[3]
tgt	VSR[XT].word[0]	VSR[XT].word[1]	VSR[XT].word[2]	VSR[XT].word[3]

0 32 64 96 127

VSR Data Layout for xvnmaddmsp

src1	VSR[XA].word[0]	VSR[XA].word[1]	VSR[XA].word[2]	VSR[XA].word[3]
src2	VSR[XB].word[0]	VSR[XB].word[1]	VSR[XB].word[2]	VSR[XB].word[3]
src3	VSR[XT].word[0]	VSR[XT].word[1]	VSR[XT].word[2]	VSR[XT].word[3]
tgt	VSR[XT].word[0]	VSR[XT].word[1]	VSR[XT].word[2]	VSR[XT].word[3]

0 32 64 96 127

Part 1: Multiply	src3: −Infinity	−NZF	−Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
src1: −Infinity	p ← +Infinity	p ← +Infinity	p ← dQNaN vximz_flag ← 1	p ← dQNaN vximz_flag ← 1	p ← −Infinity	p ← −Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
−NZF	p ← +Infinity	p ← M(src1,src3)	p ← +Zero	p ← −Zero	p ← M(src1,src3)	p ← −Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
−Zero	p ← dQNaN vximz_flag ← 1	p ← +Zero	p ← +Zero	p ← −Zero	p ← −Zero	p ← dQNaN vximz_flag ← 1	p ← src3	p ← Q(src3) vxsnan_flag ← 1
+Zero	p ← dQNaN vximz_flag ← 1	p ← −Zero	p ← −Zero	p ← +Zero	p ← +Zero	p ← dQNaN vximz_flag ← 1	p ← src3	p ← Q(src3) vxsnan_flag ← 1
+NZF	p ← −Infinity	p ← M(src1,src3)	p ← −Zero	p ← +Zero	p ← M(src1,src3)	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
+Infinity	p ← −Infinity	p ← −Infinity	p ← dQNaN vximz_flag ← 1	p ← dQNaN vximz_flag ← 1	p ← +Infinity	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
QNaN	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1 vxsnan_flag ← 1
SNaN	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1

Part 2: Add	src2: −Infinity	−NZF	−Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
p: −Infinity	v ← −Infinity	v ← −Infinity	v ← −Infinity	v ← −Infinity	v ← −Infinity	v ← dQNaN vxisi_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
−NZF	v ← −Infinity	v ← A(p,src2)	v ← p	v ← p	v ← A(p,src2)	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
−Zero	v ← −Infinity	v ← src2	v ← −Zero	v ← Rezd	v ← src2	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Zero	v ← −Infinity	v ← src2	v ← Rezd	v ← +Zero	v ← src2	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+NZF	v ← −Infinity	v ← A(p,src2)	v ← p	v ← p	v ← A(p,src2)	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Infinity	v ← dQNaN vxisi_flag ← 1	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
QNaN & src1 is a NaN	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p vxsnan_flag ← 1
QNaN & src1 not a NaN	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← src2	v ← Q(src2) vxsnan_flag ← 1

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}). For ***xvnmaddmsp***, the single-precision floating-point value in word element i of VSR[XB] (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}). For ***xvnmaddmsp***, the single-precision floating-point value in word element i of VSR[XT] (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.

Table 7.55: Actions for xvnmadd(a|m)sp

VSX Vector Negative Multiply-Subtract Type-A Double-Precision XX3-form

xvnmsubadp XT,XA,XB

60	T	A	B	241	AX	BX	TX
0	6	11	16	21	29	30	31

VSX Vector Negative Multiply-Subtract Type-M Double-Precision XX3-form

xvnmsubmdp XT,XA,XB

60	T	A	B	249	AX	BX	TX
0	6	11	16	21	29	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

ex_flag ← 0b0

do i = 0 to 1
  reset_xflags()

  if ‘‘xvnmsubadp’’ then do
    src1 ← bfp_CONVERT_FROM_BFP64(VSR[32×AX+A].dword[i])
    src2 ← bfp_CONVERT_FROM_BFP64(VSR[32×TX+T].dword[i])
    src3 ← bfp_CONVERT_FROM_BFP64(VSR[32×BX+B].dword[i])
  end
  else do
    src1 ← bfp_CONVERT_FROM_BFP64(VSR[32×AX+A].dword[i])
    src2 ← bfp_CONVERT_FROM_BFP64(VSR[32×BX+B].dword[i])
    src3 ← bfp_CONVERT_FROM_BFP64(VSR[32×TX+T].dword[i])
  end

  v   ← bfp_MULTIPLY_ADD(src1,src3,bfp_NEGATE(src2))
  rnd ← bfp_NEGATE(bfp_ROUND_TO_BFP64(FPSCR.RN,v))

  vresult.dword[i] ← bfp64_CONVERT_FROM_BFP(rnd)

  if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
  if vximz_flag=1 then SetFX(FPSCR.VXIMZ)
  if vxisi_flag=1 then SetFX(FPSCR.VXISI)
  if ox_flag=1 then SetFX(FPSCR.OX)
  if ux_flag=1 then SetFX(FPSCR.UX)
  if xx_flag=1 then SetFX(FPSCR.XX)

  ex_flag ← ex_flag | (FPSCR.VE & vxsnan_flag)
                    | (FPSCR.VE & vximz_flag)
                    | (FPSCR.VE & vxisi_flag)
                    | (FPSCR.OE & ox_flag)
                    | (FPSCR.UE & ux_flag)
                    | (FPSCR.XE & xx_flag)
end

if ex_flag=0 then VSR[32×TX+T] ← vresult
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

For each integer value i from 0 to 1, do the following.

For ***xvmsubadp***, do the following.

- Let src1 be the double-precision floating-point operand in doubleword element i of VSR[XA].
- Let src2 be the double-precision floating-point operand in doubleword element i of VSR[XT].
- Let src3 be the double-precision floating-point operand in doubleword element i of VSR[XB].

For ***xvmsubmdp***, do the following.

- Let src1 be the double-precision floating-point operand in doubleword element i of VSR[XA].
- Let src2 be the double-precision floating-point operand in doubleword element i of VSR[XB].
- Let src3 be the double-precision floating-point operand in doubleword element i of VSR[XT].

src1 is multiplied[99] by src3, producing a product having unbounded range and precision.

See part 1 of Table 7.56.

src2 is negated and added[100] to the product, producing a sum having unbounded range and precision.

The sum is normalized[101].

See part 2 of Table 7.56.

The intermediate result is rounded to double-precision using the rounding mode specified by RN.

See Table 7.9, “Scalar Floating-Point Intermediate Result Handling,” on page 617.

The result is negated and placed into doubleword element i of VSR[XT] in double-precision format.

See Table 7.54, “Vector Floating-Point Final Result with Negation,” on page 725.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered:

Register	*Field(s)*
FPSCR	FX OX UX XX VXSNAN VXISI VXIMZ

[99] Floating-point multiplication is based on exponent addition and multiplication of the significands.
[100] 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.
[101] 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 xvnmsubadp

src1	VSR[XA].dword[0]	VSR[XA].dword[1]
src2	VSR[XT].dword[0]	VSR[XT].dword[1]
src3	VSR[XB].dword[0]	VSR[XB].dword[1]
tgt	VSR[XT].dword[0]	VSR[XT].dword[1]
	0	64 127

VSR Data Layout for xvnmsubmdp

src1	VSR[XA].dword[0]	VSR[XA].dword[1]
src2	VSR[XB].dword[0]	VSR[XB].dword[1]
src3	VSR[XT].dword[0]	VSR[XT].dword[1]
tgt	VSR[XT].dword[0]	VSR[XT].dword[1]
	0	64 127

Part 1: Multiply

src1 \ src3	−Infinity	−NZF	−Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
−Infinity	p ← +Infinity	p ← +Infinity	p ← dQNaN vximz_flag ← 1	p ← dQNaN vximz_flag ← 1	p ← −Infinity	p ← −Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
−NZF	p ← +Infinity	p ← M(src1,src3)	p ← +Zero	p ← −Zero	p ← M(src1,src3)	p ← −Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
−Zero	p ← dQNaN vximz_flag ← 1	p ← +Zero	p ← +Zero	p ← −Zero	p ← −Zero	p ← dQNaN vximz_flag ← 1	p ← src3	p ← Q(src3) vxsnan_flag ← 1
+Zero	p ← dQNaN vximz_flag ← 1	p ← −Zero	p ← −Zero	p ← +Zero	p ← +Zero	p ← dQNaN vximz_flag ← 1	p ← src3	p ← Q(src3) vxsnan_flag ← 1
+NZF	p ← −Infinity	p ← M(src1,src3)	p ← −Zero	p ← +Zero	p ← M(src1,src3)	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
+Infinity	p ← −Infinity	p ← −Infinity	p ← dQNaN vximz_flag ← 1	p ← dQNaN vximz_flag ← 1	p ← +Infinity	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
QNaN	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1 vxsnan_flag ← 1
SNaN	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1

Part 2: Subtract

p \ src2	−Infinity	−NZF	−Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
−Infinity	v ← dQNaN vxisi_flag ← 1	v ← −Infinity	v ← −Infinity	v ← −Infinity	v ← −Infinity	v ← −Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
−NZF	v ← +Infinity	v ← S(p,src2)	v ← p	v ← p	v ← S(p,src2)	v ← −Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
−Zero	v ← +Infinity	v ← −src2	v ← Rezd	v ← −Zero	v ← −src2	v ← −Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Zero	v ← +Infinity	v ← −src2	v ← +Zero	v ← Rezd	v ← −src2	v ← −Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+NZF	v ← +Infinity	v ← S(p,src2)	v ← p	v ← p	v ← S(p,src2)	v ← −Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← dQNaN vxisi_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
QNaN & src1 is a NaN	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p vxsnan_flag ← 1
QNaN & src1 not a NaN	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← src2	v ← Q(src2) vxsnan_flag ← 1

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 i of VSR[XT] (where i={0,1}). For ***xvnmsubmdp***, the double-precision floating-point value in doubleword element i of VSR[XB] (where i={0,1}).
src3	For ***xvnmsubadp***, the double-precision floating-point value in doubleword element i of VSR[XB] (where i={0,1}). For ***xvnmsubmdp***, the double-precision floating-point value in doubleword element i of VSR[XT] (where i={0,1}).
dQNaN	Default quiet NaN (`0x7FF8_0000_0000_0000`).
NZF	Nonzero finite number.
Rezd	Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs). Can also occur with two nonzero finite number source operands.
Q(x)	Return a QNaN with the payload of x.
S(x,y)	Return the normalized sum of floating-point value x and negated floating-point value y, having unbounded range and precision. Note: If x = -y, v is considered to be an exact-zero-difference result (Rezd).
M(x,y)	Return the normalized product of floating-point value x and floating-point value y, having unbounded range and precision.
p	The intermediate product having unbounded range and precision.
v	The intermediate result having unbounded range and precision.

Table 7.56: Actions for xvnmsub(a|m)dp

VSX Vector Negative Multiply-Subtract Type-A Single-Precision XX3-form

xvnmsubasp XT,XA,XB

60	T	A	B	209	AX	BX	TX
0	6	11	16	21	29	30	31

VSX Vector Negative Multiply-Subtract Type-M Single-Precision XX3-form

xvnmsubmsp XT,XA,XB

60	T	A	B	217	AX	BX	TX
0	6	11	16	21	29	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

ex_flag ← 0b0

do i = 0 to 3
  reset_xflags()

  if ‘‘xvnmsubasp’’ then do
    src1 ← bfp_CONVERT_FROM_BFP32(VSR[32×AX+A].word[i])
    src2 ← bfp_CONVERT_FROM_BFP32(VSR[32×TX+T].word[i])
    src3 ← bfp_CONVERT_FROM_BFP32(VSR[32×BX+B].word[i])
  end
  else do
    src1 ← bfp_CONVERT_FROM_BFP32(VSR[32×AX+A].word[i])
    src2 ← bfp_CONVERT_FROM_BFP32(VSR[32×BX+B].word[i])
    src3 ← bfp_CONVERT_FROM_BFP32(VSR[32×TX+T].word[i])
  end

  v   ← bfp_MULTIPLY_ADD(src1,src3,bfp_NEGATE(src2))
  rnd ← bfp_NEGATE(bfp_ROUND_TO_BFP32(FPSCR.RN,v))

  vresult.word[i] ← bfp32_CONVERT_FROM_BFP(rnd)

  if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
  if vximz_flag=1 then SetFX(FPSCR.VXIMZ)
  if vxisi_flag=1 then SetFX(FPSCR.VXISI)
  if ox_flag=1 then SetFX(FPSCR.OX)
  if ux_flag=1 then SetFX(FPSCR.UX)
  if xx_flag=1 then SetFX(FPSCR.XX)

  ex_flag ← ex_flag | (FPSCR.VE & vxsnan_flag)
                    | (FPSCR.VE & vximz_flag)
                    | (FPSCR.VE & vxisi_flag)
                    | (FPSCR.OE & ox_flag)
                    | (FPSCR.UE & ux_flag)
                    | (FPSCR.XE & xx_flag)
end

if ex_flag=0 then VSR[32×TX+T] ← vresult
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

For each integer value i from 0 to 3, do the following.

For ***xvnmsubasp***, do the following.

- Let src1 be the single-precision floating-point operand in word element i of VSR[XA].
- Let src2 be the single-precision floating-point operand in word element i of VSR[XT].
- Let src3 be the single-precision floating-point operand in word element i of VSR[XB].

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 multiplied[102] by src3, producing a product having unbounded range and precision.

See part 1 of Table 7.57.

src2 is negated and added[103] to the product, producing a sum having unbounded range and precision.

The sum is normalized[104].

See part 2 of Table 7.57.

The intermediate result is rounded to single-precision using the rounding mode specified by RN.

See Table 7.9, “Scalar Floating-Point Intermediate Result Handling,” on page 617.

The result is negated and placed into word element i of VSR[XT] in single-precision format.

See Table 7.54, “Vector Floating-Point Final Result with Negation,” on page 725.

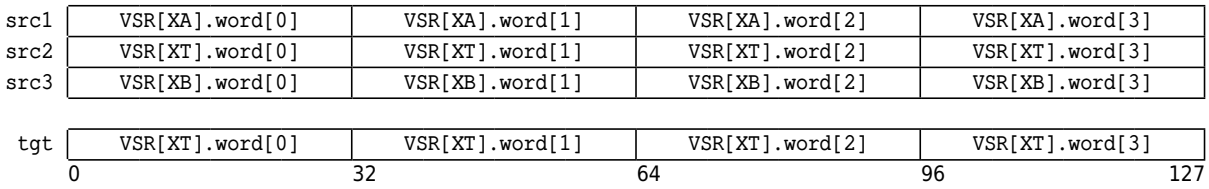
If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered:

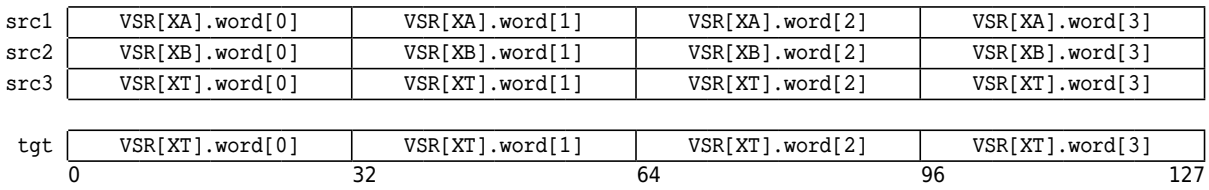
Register	*Field(s)*
FPSCR	FX OX UX XX VXSNAN VXISI VXIMZ

[102] Floating-point multiplication is based on exponent addition and multiplication of the significands.
[103] 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.
[104] Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.

VSR Data Layout for xvnmsubasp



VSR Data Layout for xvnmsubmsp



Part 1: Multiply	src3							
src1	−Infinity	−NZF	−Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
−Infinity	p ← +Infinity	p ← +Infinity	p ← dQNaN vximz_flag ← 1	p ← dQNaN vximz_flag ← 1	p ← −Infinity	p ← −Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
−NZF	p ← +Infinity	p ← M(src1,src3)	p ← +Zero	p ← −Zero	p ← M(src1,src3)	p ← −Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
−Zero	p ← dQNaN vximz_flag ← 1	p ← +Zero	p ← +Zero	p ← −Zero	p ← −Zero	p ← dQNaN vximz_flag ← 1	p ← src3	p ← Q(src3) vxsnan_flag ← 1
+Zero	p ← dQNaN vximz_flag ← 1	p ← −Zero	p ← −Zero	p ← +Zero	p ← +Zero	p ← dQNaN vximz_flag ← 1	p ← src3	p ← Q(src3) vxsnan_flag ← 1
+NZF	p ← −Infinity	p ← M(src1,src3)	p ← −Zero	p ← +Zero	p ← M(src1,src3)	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
+Infinity	p ← −Infinity	p ← −Infinity	p ← dQNaN vximz_flag ← 1	p ← dQNaN vximz_flag ← 1	p ← +Infinity	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
QNaN	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1 vxsnan_flag ← 1
SNaN	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1

Part 2: Subtract	src2							
p	−Infinity	−NZF	−Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
−Infinity	v ← dQNaN vxisi_flag ← 1	v ← −Infinity	v ← −Infinity	v ← −Infinity	v ← −Infinity	v ← −Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
−NZF	v ← +Infinity	v ← S(p,src2)	v ← p	v ← p	v ← S(p,src2)	v ← −Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
−Zero	v ← +Infinity	v ← −src2	v ← Rezd	v ← −Zero	v ← −src2	v ← −Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Zero	v ← +Infinity	v ← −src2	v ← +Zero	v ← Rezd	v ← −src2	v ← −Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+NZF	v ← +Infinity	v ← S(p,src2)	v ← p	v ← p	v ← S(p,src2)	v ← −Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← dQNaN vxisi_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
QNaN & src1 is a NaN	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p vxsnan_flag ← 1
QNaN & src1 not a NaN	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← src2	v ← Q(src2) vxsnan_flag ← 1

Explanation:

src1	The single-precision floating-point value in word element i of VSR[XA] (where i={0,1,2,3}).
src2	For ***xvnmsubasp***, the single-precision floating-point value in word element i of VSR[XT] (where i={0,1,2,3}). For ***xvnmsubmsp***, the single-precision floating-point value in word element i of VSR[XB] (where i={0,1,2,3}).
src3	For ***xvnmsubasp***, the single-precision floating-point value in word element i of VSR[XB] (where i={0,1,2,3}). For ***xvnmsubmsp***, the single-precision floating-point value in word element i of VSR[XT] (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. 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 7.57: Actions for xvnmsub(a|m)sp

VSX Vector Reciprocal Estimate Double-Precision XX2-form

xvredp XT,XB

60	T	///	B	218	BX	TX
0	6	11	16	21	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

ex_flag ← 0b0

do i = 0 to 1
   reset_xflags()

   src ← bfp_CONVERT_FROM_BFP64(VSR[32×BX+B].dword[i])
   v   ← bfp_RECIPROCAL_ESTIMATE(src)
   rnd ← bfp_ROUND_TO_BFP64(0b0,FPSCR.RN,v)

   vresult.word[i] ← bfp64_CONVERT_FROM_BFP(rnd)

   if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
   if ox_flag=1 then SetFX(FPSCR.OX)
   if ux_flag=1 then SetFX(FPSCR.UX)
   if zx_flag=1 then SetFX(FPSCR.ZX)

   ex_flag ← ex_flag | (FPSCR.VE & vxsnan_flag)
                     | (FPSCR.OE & ox_flag)
                     | (FPSCR.UE & ux_flag)
                     | (FPSCR.ZE & zx_flag)
end

if ex_flag=0 then VSR[32×TX+T] ← vresult
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

For each integer value i from 0 to 1, do the following.

Let src be the double-precision floating-point operand in doubleword element i of VSR[XB].

A double-precision floating-point estimate of the reciprocal of src is placed into doubleword element i of VSR[XT] in double-precision format.

Unless the reciprocal of src would be a zero, an infinity, or a QNaN, the estimate has a relative error in precision no greater than one part in 16384 of the reciprocal of src. That is,

$$\left| \frac{\text{estimate} - \frac{1}{\text{src}}}{\frac{1}{\text{src}}} \right| \leq \frac{1}{16384}$$

Operation with various special values of the operand is summarized below.

Source Value	Result	Exception
−Infinity	−Zero	None
−Zero	−Infinity[1]	ZX
+Zero	+Infinity[1]	ZX
+Infinity	+Zero	None
SNaN	QNaN[2]	VXSNAN
QNaN	QNaN	None

[1] No result if VE=1.
[2] No result if ZE=1.

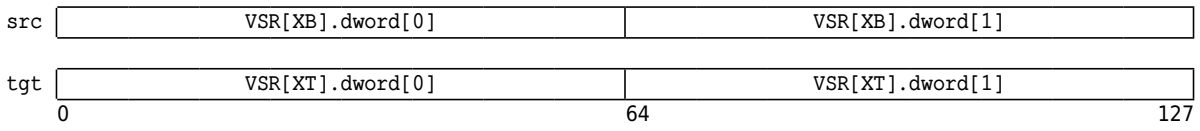
If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

The results of executing this instruction is permitted to vary between implementations, and between different executions on the same implementation.

Special Registers Altered:

Register	*Field(s)*
FPSCR	FX OX UX ZX VXSNAN

VSR Data Layout for xvredp



VSX Vector Reciprocal Estimate Single-Precision XX2-form

xvresp XT,XB

60	T	///	B	154	BX	TX
0	6	11	16	21	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

ex_flag ← 0b0

do i = 0 to 3
   reset_xflags()

   src ← bfp_CONVERT_FROM_BFP32(VSR[32×BX+B].word[i])
   v   ← bfp_RECIPROCAL_ESTIMATE(src)
   rnd ← bfp_ROUND_TO_BFP32(FPSCR.RN,v)

   vresult.word[i] ← bfp32_CONVERT_FROM_BFP(rnd)

   if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
   if ox_flag=1 then SetFX(FPSCR.OX)
   if ux_flag=1 then SetFX(FPSCR.UX)
   if zx_flag=1 then SetFX(FPSCR.ZX)

   ex_flag ← ex_flag | (FPSCR.VE & vxsnan_flag)
                     | (FPSCR.OE & ox_flag)
                     | (FPSCR.UE & ux_flag)
                     | (FPSCR.ZE & zx_flag)
end

if ex_flag=0 then VSR[32×TX+T] ← vresult
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

For each integer value i from 0 to 3, do the following.

Let src be the single-precision floating-point operand in word element i of VSR[XB].

A single-precision floating-point estimate of the reciprocal 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.

Source Value	Result	Exception
−Infinity	−Zero	None
−Zero	−Infinity[1]	ZX
+Zero	+Infinity[1]	ZX
+Infinity	+Zero	None
SNaN	QNaN[2]	VXSNAN
QNaN	QNaN	None

[1] No result if VE=1.
[2] No result if ZE=1.

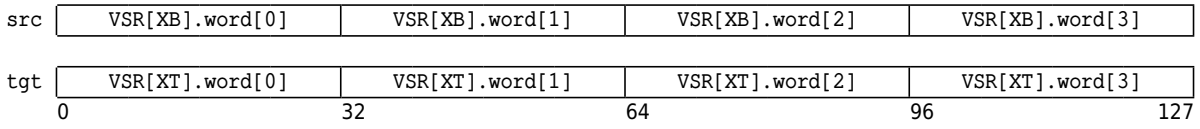
If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

The results of executing this instruction is permitted to vary between implementations, and between different executions on the same implementation.

Special Registers Altered:

Register	*Field(s)*
FPSCR	FX OX UX ZX VXSNAN

VSR Data Layout for xvresp



VSX Vector Reciprocal Square Root Estimate Double-Precision XX2-form

xvrsqrtedp XT,XB

60	T	///	B	202	BX	TX
0	6	11	16	21	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

ex_flag ← 0b0

do i = 0 to 1
   reset_xflags()

   src ← bfp_CONVERT_FROM_BFP64(VSR[32×BX+B].dword[i])
   v   ← bfp_RECIPROCAL_SQUARE_ROOT_ESTIMATE(src)
   rnd ← bfp_ROUND_TO_BFP64(0b0,FPSCR.RN,v)

   vresult.dword[i] ← bfp64_CONVERT_FROM_BFP(rnd)

   if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
   if vxsqrt_flag=1 then SetFX(FPSCR.VXSQRT)
   if zx_flag=1 then SetFX(FPSCR.ZX)

   ex_flag ← ex_flag | (FPSCR.VE & vxsnan_flag)
                     | (FPSCR.VE & vxsqrt_flag)
                     | (FPSCR.ZE & zx_flag)
end

if ex_flag=0 then VSR[32×TX+T] ← vresult
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

For each integer value i from 0 to 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,

$$\left| \frac{\texttt{estimate} - \frac{1}{\sqrt{\texttt{src}}}}{\frac{1}{\sqrt{\texttt{src}}}} \right| \leq \frac{1}{16384}$$

Operation with various special values of the operand is summarized below.

Source Value	Result	Exception
−Infinity	QNaN[1]	VXSQRT
+Infinity	+Zero	None
−Finite	QNaN[1]	VXSQRT
−Zero	−Infinity[2]	ZX
+Zero	+Infinity[2]	ZX
SNaN	QNaN[1]	VXSNAN
QNaN	QNaN	None

[1] No result if VE=1.
[2] No result if ZE=1.

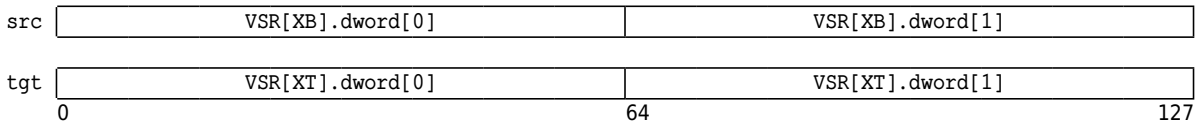
If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

The results of executing this instruction is permitted to vary between implementations, and between different executions on the same implementation.

Special Registers Altered:

Register	*Field(s)*
FPSCR	FX ZX VXSNAN VXSQRT

VSR Data Layout for xvrsqrtedp



VSX Vector Reciprocal Square Root Estimate Single-Precision XX2-form

xvrsqrtesp XT,XB

60	T	///	B	138	BX	TX
0	6	11	16	21	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

ex_flag ← 0b0

do i = 0 to 3
   reset_xflags()

   src ← bfp_CONVERT_FROM_BFP32(VSR[32×BX+B].word[i])
   v   ← bfp_RECIPROCAL_SQUARE_ROOT_ESTIMATE(src)
   rnd ← bfp_ROUND_TO_BFP32(FPSCR.RN,v)

   vresult.word[i] ← bfp32_CONVERT_FROM_BFP(rnd)

   if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
   if vxsqrt_flag=1 then SetFX(FPSCR.VXSQRT)
   if zx_flag=1 then SetFX(FPSCR.ZX)

   ex_flag ← ex_flag | (FPSCR.VE & vxsnan_flag)
                     | (FPSCR.VE & vxsqrt_flag)
                     | (FPSCR.ZE & zx_flag)
end

if ex_flag=0 then VSR[32×TX+T] ← vresult
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

For each integer value i from 0 to 3, do the following.

Let src be the single-precision floating-point operand in word element i of VSR[XB].

A single-precision floating-point estimate of the reciprocal square root of src is placed into word element i of VSR[XT] in single-precision format.

Unless the reciprocal of the square root of src would be a zero, an infinity, or a QNaN, the estimate has a relative error in precision no greater than one part in 16384 of the reciprocal of the square root of src. That is,

$$\left| \frac{\texttt{estimate} - \frac{1}{\sqrt{\texttt{src}}}}{\frac{1}{\sqrt{\texttt{src}}}} \right| \leq \frac{1}{16384}$$

Operation with various special values of the operand is summarized below.

Source Value	Result	Exception
−Infinity	QNaN[1]	VXSQRT
+Infinity	+Zero	None
−Finite	QNaN[1]	VXSQRT
−Zero	−Infinity[2]	ZX
+Zero	+Infinity[2]	ZX
SNaN	QNaN[1]	VXSNAN
QNaN	QNaN	None

[1] No result if VE=1.
[2] No result if ZE=1.

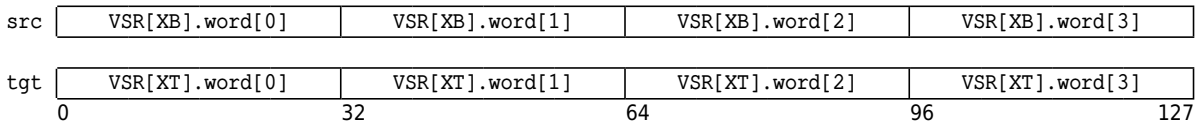
If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

The results of executing this instruction is permitted to vary between implementations, and between different executions on the same implementation.

Special Registers Altered:

Register	*Field(s)*
FPSCR	FX ZX VXSNAN VXSQRT

VSR Data Layout for xvrsqrtesp



VSX Vector Test for software Divide Double-Precision XX3-form

xvtdivdp BF,XA,XB

60	BF	//	A	B	125	AX	BX	/
0	6	9	11	16	21	29	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

eq_flag ← 0b0
gt_flag ← 0b0

do i = 0 to 1
   src1    ← VSR[32×AX+A].dword[i]
   src2    ← VSR[32×BX+B].dword[i]
   e_a     ← src1.bit[1:11] - 1023
   e_b     ← src2.bit[1:11] - 1023
   fe_flag ← fe_flag |
             IsNaN(src1) | IsInf(src1) |
             IsNaN(src2) | IsInf(src2) |
             IsZero(src2) |
             (e_b <= -1022) |
             (e_b >=  1021) |
             (!IsZero(src1) & ((e_a - e_b) >=  1023))
|
             (!IsZero(src1) & ((e_a - e_b) <= -1021))
|
             (!IsZero(src1) & (e_a <= -970))
   fg_flag ← fg_flag |
             IsInf(src1) | IsInf(src2) |
             IsZero(src2) | IsDen(src2)
end

fl_flag ← xvredp_error() <= 2^-14
CR[BF]  ← 0b1 || fg_flag || fe_flag || 0b0
```

Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

fe_flag is initialized to 0.
fg_flag is initialized to 0.

For each integer value i from 0 to 1, do the following.

Let src1 be the double-precision floating-point operand in doubleword element i of VSR[XA].
Let src2 be the double-precision floating-point operand in doubleword element i of VSR[XB].

Let e_a be the unbiased exponent of src1.
Let e_b be the unbiased exponent of src2.

fe_flag is set to 1 for any of the following conditions.

- src1 is a NaN or an infinity.
- src2 is a zero, a NaN, or an infinity.
- e_b is less than or equal to -1022.
- e_b is greater than or equal to 1021.
- src1 is not a zero and the difference, e_a - e_b, is greater than or equal to 1023.
- src1 is not a zero and the difference, e_a - e_b, is less than or equal to -1021.
- src1 is not a zero and e_a is less than or equal to -970

fg_flag is set to 1 for any of the following conditions.

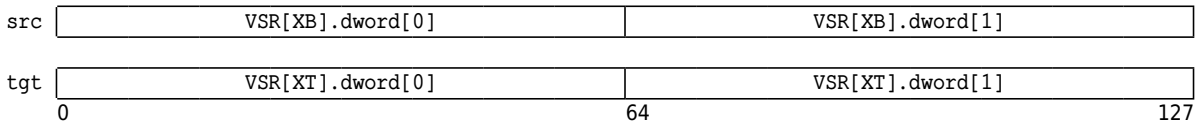
- src1 is an infinity.
- src2 is a zero, an infinity, or a denormalized value.

CR field BF is set to the value 0b1 || fg_flag || fe_flag || 0b0.

Special Registers Altered:

Register	*Field(s)*
CR	field BF

VSR Data Layout for xvtdivdp



VSX Vector Test for software Divide Single-Precision XX3-form

xvtdivsp BF,XA,XB

60	BF	//	A	B	93	AX	BX	/
0	6	9	11	16	21	29	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

eq_flag ← 0b0
gt_flag ← 0b0

do i = 0 to 3
   src1    ← VSR[32×AX+A].word[i]
   src2    ← VSR[32×BX+B].word[i]
   e_a     ← src1.bit[1:8] - 127
   e_b     ← src2.bit[1:8] - 127
   fe_flag ← fe_flag |
             IsNaN(src1) | IsInf(src1) |
             IsNaN(src2) | IsInf(src2) |
             IsZero(src2) |
             (e_b <= -126) |
             (e_b >=  125) |
             (!IsZero(src1) & ((e_a - e_b) >=  127)) |
             (!IsZero(src1) & ((e_a - e_b) <= -125)) |
             (!IsZero(src1) & (e_a <= -103))
   fg_flag ← fg_flag |
             IsInf(src1) | IsInf(src2) |
             IsZero(src2) | IsDen(src2)
end

fl_flag ← xvredp_error() <= 2^-14
CR.field[BF]  ← 0b1 || fg_flag || fe_flag || 0b0
```

Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

fe_flag is initialized to 0.
fg_flag is initialized to 0.

For each integer value i from 0 to 3, do the following.

Let src1 be the single-precision floating-point operand in word element i of VSR[XA].
Let src2 be the single-precision floating-point operand in word element i of VSR[XB].

Let e_a be the unbiased exponent of src1.
Let e_b be the unbiased exponent of src2.

fe_flag is set to 1 for any of the following conditions.

- src1 is a NaN or an infinity.
- src2 is a zero, a NaN, or an infinity.
- e_b is less than or equal to −126.
- e_b is greater than or equal to 125.
- src1 is not a zero and the difference, e_a − e_b, is greater than or equal to 127.
- src1 is not a zero and the difference, e_a − e_b, is less than or equal to −125.
- src1 is not a zero and e_a is less than or equal to −103.

fg_flag is set to 1 for any of the following conditions.

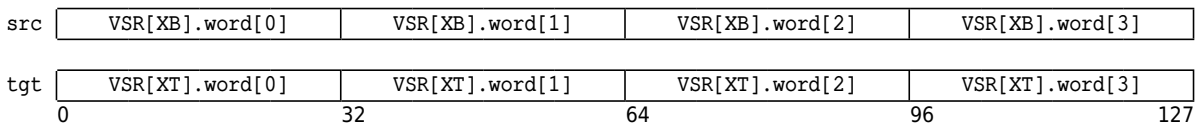
- src1 is an infinity.
- src2 is a zero, an infinity, or a denormalized value.

CR field BF is set to the value 0b1 || fg_flag || fe_flag || 0b0.

Special Registers Altered:

Register	*Field(s)*
CR	field BF

VSR Data Layout for xvtdivsp



VSX Vector Test for software Square Root Double-Precision XX2-form

xvtsqrtdp BF,XB

60	BF	//	///	B	234	BX	/
0	6	9	11	16	21	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

fe_flag ← 0b0
fg_flag ← 0b0

do i = 0 to 1
   src     ← VSR[32×BX+B].dword[i]
   e_b     ← src2.bit[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.field[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 integer value i from 0 to 1, do the following.

Let src be the double-precision floating-point operand in doubleword element i of VSR[XB].

Let e_b be the unbiased exponent of src.

fe_flag is set to 1 for any of the following conditions.

- src is a zero, a NaN, an infinity, or a negative value.
- e_b is less than or equal to -970.

fg_flag is set to 1 for the following condition.

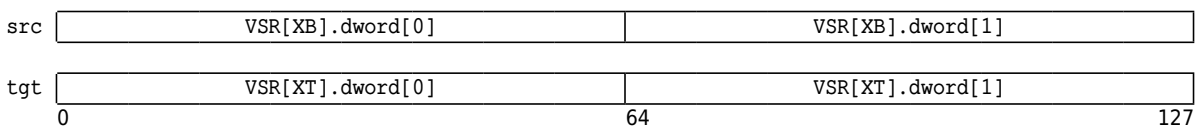
- src is a zero, an infinity, or a denormalized value.

CR field BF is set to the value 0b1 || fg_flag || fe_flag || 0b0.

Special Registers Altered:

Register	*Field(s)*
CR	field BF

VSR Data Layout for xvtsqrtdp



VSX Vector Test for software Square Root Single-Precision XX2-form

xvtsqrtsp BF,XB

60	BF	//	///	B	170	BX	/
0	6	9	11	16	21	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

fe_flag ← 0b0
fg_flag ← 0b0

do i = 0 to 3
   src     ← VSR[32×BX+B].word[i]
   e_b     ← src2.bit[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.field[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 integer value i from 0 to 3, do the following.

Let src be the single-precision floating-point operand in word element i of VSR[XB].

Let e_b be the unbiased exponent of src.

fe_flag is set to 1 for any of the following conditions.

- src is a zero, a NaN, an infinity, or a negative value.
- e_b is less than or equal to -103.

fg_flag is set to 1 for the following condition.

- src is a zero, an infinity, or a denormalized value.

CR field BF is set to the value 0b1 || fg_flag || fe_flag || 0b0.

Special Registers Altered:

Register	*Field(s)*
CR	field BF

VSR Data Layout for xvtsqrtsp

7.6.2.4 VSX Binary Floating-Point Compare Instructions

7.6.2.4.1 VSX Scalar BFP Compare Instructions

VSX Scalar Compare Ordered Double-Precision XX3-form

xscmpodp BF,XA,XB

60	BF	//	A	B	43	AX	BX	/
0	6	9	11	16	21	29	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

reset_xflags()

src1 ← bfp_CONVERT_FROM_BFP64(VSR[32×AX+A].dword[0])
src2 ← bfp_CONVERT_FROM_BFP64(VSR[32×BX+B].dword[0])

if src1.class.SNaN=1 | src2.class.SNaN=1 then do
   vxsnan_flag ← 0b1
   if FPSCR.VE=0 then vxvc_flag ← 0b1
end
else
   vxvc_flag ← src1.class.QNaN | src2.class.QNaN

if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
if vxvc_flag=1   then SetFX(FPSCR.VXVC)

CR.bit[4×BF+32] ← FPSCR.FL ← src1 < src2
CR.bit[4×BF+33] ← FPSCR.FG ← src1 > src2
CR.bit[4×BF+34] ← FPSCR.FE ← src1 = src2
CR.bit[4×BF+35] ← FPSCR.FU ←
   src1.class.SNaN | src1.class.QNaN |
   src2.class.SNaN | src2.class.QNaN
```

Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

Let src1 be the double-precision floating-point value in doubleword element 0 of VSR[XA].

Let src2 be the double-precision floating-point value in doubleword element 0 of VSR[XB].

src1 is compared to src2.

Zeros of same or opposite signs compare equal.

Infinities of same signs compare equal.

See Table 7.58, “Actions for xscmpodp - Part 1: Compare Ordered,” on page 744.

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 7.59, “Actions for xscmpodp - Part 2: Result,” on page 744.

Special Registers Altered:

Register	*Field(s)*
CR	field BF
FPSCR	FPCC FX VXSNAN VXVC

Programming Note

This instruction can be used to operate on single-precision source operands.

VSR Data Layout for xscmpodp

src1	VSR[XA].dword[0]	unused
src2	VSR[XB].dword[0]	unused
	0	64 127

src1 \ src2	−Infinity	−NZF	−Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
−Infinity	cc←0b0010	cc←0b1000	cc←0b1000	cc←0b1000	cc←0b1000	cc←0b1000	cc←0b0001 vxvc_flag←1	cc←0b0001 vxsnan_flag←1 vxvc_flag←(VE=0)
−NZF	cc←0b0100	cc←C(src1,src2)	cc←0b1000	cc←0b1000	cc←0b1000	cc←0b1000	cc←0b0001 vxvc_flag←1	cc←0b0001 vxsnan_flag←1 vxvc_flag←(VE=0)
−Zero	cc←0b0100	cc←0b0100	cc←0b0010	cc←0b0010	cc←0b1000	cc←0b1000	cc←0b0001 vxvc_flag←1	cc←0b0001 vxsnan_flag←1 vxvc_flag←(VE=0)
+Zero	cc←0b0100	cc←0b0100	cc←0b0010	cc←0b0010	cc←0b1000	cc←0b1000	cc←0b0001 vxvc_flag←1	cc←0b0001 vxsnan_flag←1 vxvc_flag←(VE=0)
+NZF	cc←0b0100	cc←0b0100	cc←0b0100	cc←0b0100	cc←C(src1,src2)	cc←0b1000	cc←0b0001 vxvc_flag←1	cc←0b0001 vxsnan_flag←1 vxvc_flag←(VE=0)
+Infinity	cc←0b0100	cc←0b0100	cc←0b0100	cc←0b0100	cc←0b0100	cc←0b0010	cc←0b0001 vxvc_flag←1	cc←0b0001 vxsnan_flag←1 vxvc_flag←(VE=0)
QNaN	cc←0b0001 vxvc_flag←1	cc←0b0001 vxvc_flag←1	cc←0b0001 vxvc_flag←1	cc←0b0001 vxvc_flag←1	cc←0b0001 vxvc_flag←1	cc←0b0001 vxvc_flag←1	cc←0b0001 vxvc_flag←1	cc←0b0001 vxsnan_flag←1 vxvc_flag←(VE=0)
SNaN	cc←0b0001 vxsnan_flag←1 vxvc_flag←(VE=0)	cc←0b0001 vxsnan_flag←1 vxvc_flag←(VE=0)	cc←0b0001 vxsnan_flag←1 vxvc_flag←(VE=0)	cc←0b0001 vxsnan_flag←1 vxvc_flag←(VE=0)	cc←0b0001 vxsnan_flag←1 vxvc_flag←(VE=0)	cc←0b0001 vxsnan_flag←1 vxvc_flag←(VE=0)	cc←0b0001 vxsnan_flag←1 vxvc_flag←(VE=0)	cc←0b0001 vxsnan_flag←1 vxvc_flag←(VE=0)

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 7.58: Actions for xscmpodp - Part 1: Compare Ordered

VE	vxsnan_flag	vxvc_flag	Returned Results and Status Setting
-	0	0	`FPCC←cc, CR[BF]←cc`
0	0	1	`FPCC←cc, CR[BF]←cc, fx(VXVC)`
0	1	0	`FPCC←cc, CR[BF]←cc, fx(VXSNAN)`
0	1	1	`FPCC←cc, CR[BF]←cc, fx(VXSNAN), fx(VXVC)`
1	0	1	`FPCC←cc, CR[BF]←cc, fx(VXVC), error()`
1	1	-	`FPCC←cc, CR[BF]←cc, fx(VXSNAN), error()`

Explanation:

-	The results do not depend on this condition.
`cc`	The 4-bit result as defined in Table 7.58.
`error()`	The system error handler is invoked for the trap-enabled exception if `MSR.FE0` and `MSR.FE1` are set to any mode other than the ignore-exception mode.
`fx(x)`	`FPSCR.FX` is set to 1 if `FPSCR.x`=0. `FPSCR.x` is set to 1.

Table 7.59: Actions for xscmpodp - Part 2: Result

VSX Scalar Compare Ordered Quad-Precision X-form

xscmpoqp BF,VRA,VRB

63	BF	//	VRA	VRB	132	/
0	6	9	11	16	21	31

```
if MSR.VSX=0 then VSX_Unavailable()

reset_xflags()

src1 ← bfp_CONVERT_FROM_BFP128(VSR[VRA+32])
src2 ← bfp_CONVERT_FROM_BFP128(VSR[VRB+32])

if src1.class.SNaN=1 | src2.class.SNaN=1 then do
   vxsnan_flag ← 0b1
   if FPSCR.VE=0 then vxvc_flag ← 0b1
end
else
   vxvc_flag ← src1.class.QNaN | src2.class.QNaN

if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
if vxvc_flag=1   then SetFX(FPSCR.VXVC)

CR.bit[4×BF+32] ← FPSCR.FL ← src1 < src2
CR.bit[4×BF+33] ← FPSCR.FG ← src1 > src2
CR.bit[4×BF+34] ← FPSCR.FE ← src1 = src2
CR.bit[4×BF+35] ← FPSCR.FU ←
   src1.class.SNaN | src1.class.QNaN |
   src2.class.SNaN | src2.class.QNaN
```

Let src1 be the floating-point value in VSR[VRA+32] represented in quad-precision format.

Let src2 be the floating-point value in VSR[VRB+32] represented in quad-precision format.

src1 is compared to src2.

Zeros of same or opposite signs compare equal. Infinities of same signs compare equal.

Bit 0 of CR field BF and FL are set to indicate if src1 is less than src2.

Bit 1 of CR field BF and FG are set to indicate if src1 is greater than src2.

Bit 2 of CR field BF and FE are set to indicate if src1 is equal to src2.

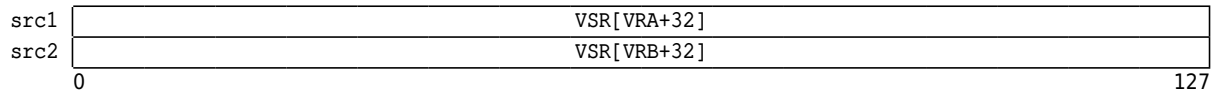
Bit 3 of CR field BF and FU are set to indicate unordered (i.e., src1 or src2 is a NaN).

If either of the operands is a 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:

Register	*Field(s)*
CR	field BF
FPSCR	FPCC FX VXSNAN VXVC

VSR Data Layout for xscmpoqp



VSX Scalar Compare Unordered Double-Precision XX3-form

xscmpudp BF,XA,XB

60	BF	//	A	B	35	AX	BX	/
0	6	9	11	16	21	29	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

reset_xflags()

src1 ← bfp_CONVERT_FROM_BFP64(VSR[32×AX+A].dword[0])
src2 ← bfp_CONVERT_FROM_BFP64(VSR[32×BX+B].dword[0])

vxsnan_flag ← src1.class.SNaN | src2.class.SNaN

if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)

CR.bit[4×BF+32] ← FPSCR.FL ← src1 < src2
CR.bit[4×BF+33] ← FPSCR.FG ← src1 > src2
CR.bit[4×BF+34] ← FPSCR.FE ← src1 = src2
CR.bit[4×BF+35] ← FPSCR.FU ←
   src1.class.SNaN | src1.class.QNaN |
   src2.class.SNaN | src2.class.QNaN
```

Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

Let src1 be the double-precision floating-point value in doubleword element 0 of VSR[XA].

Let src2 be the double-precision floating-point value in doubleword element 0 of VSR[XB].

src1 is compared to src2.

Zeros of same or opposite signs compare equal equal.

Infinities of same signs compare equal.

See Table 7.60, “Actions for xscmpudp - Part 1: Compare Unordered,” on page 747.

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 7.61, “Actions for xscmpudp - Part 2: Result,” on page 747.

Special Registers Altered:

Register	*Field(s)*
CR	field BF
FPSCR	FPCC FX VXSNAN

Programming Note

This instruction can be used to operate on single-precision source operands.

Programming Note

Previous versions of the architecture allowed the contents of doubleword 1 of the result register to be undefined. However, all processors that support this instruction write 0s into doubleword 1 of the result register, as is required by this version of the architecture.

VSR Data Layout for xscmpudp

src1	VSR[XA].dword[0]	unused
src2	VSR[XB].dword[0]	unused

0 64 127

		src2							
		−Infinity	**−NZF**	**−Zero**	**+Zero**	**+NZF**	**+Infinity**	**QNaN**	**SNaN**
src1	**−Infinity**	cc←0b0010	cc←0b1000	cc←0b1000	cc←0b1000	cc←0b1000	cc←0b1000	cc←0b0001	cc←0b0001 vxsnan_flag←1
	−NZF	cc←0b0100	cc←C(src1,src2)	cc←0b1000	cc←0b1000	cc←0b1000	cc←0b1000	cc←0b0001	cc←0b0001 vxsnan_flag←1
	−Zero	cc←0b0100	cc←0b0100	cc←0b0010	cc←0b0010	cc←0b1000	cc←0b1000	cc←0b0001	cc←0b0001 vxsnan_flag←1
	+Zero	cc←0b0100	cc←0b0100	cc←0b0010	cc←0b0010	cc←0b1000	cc←0b1000	cc←0b0001	cc←0b0001 vxsnan_flag←1
	+NZF	cc←0b0100	cc←0b0100	cc←0b0100	cc←0b0100	cc←C(src1,src2)	cc←0b1000	cc←0b0001	cc←0b0001 vxsnan_flag←1
	+Infinity	cc←0b0100	cc←0b0100	cc←0b0100	cc←0b0100	cc←0b0100	cc←0b0010	cc←0b0001	cc←0b0001 vxsnan_flag←1
	QNaN	cc←0b0001	cc←0b0001	cc←0b0001	cc←0b0001	cc←0b0001	cc←0b0001	cc←0b0001	cc←0b0001 vxsnan_flag←1
	SNaN	cc←0b0001 vxsnan_flag←1	cc←0b0001 vxsnan_flag←1	cc←0b0001 vxsnan_flag←1	cc←0b0001 vxsnan_flag←1	cc←0b0001 vxsnan_flag←1	cc←0b0001 vxsnan_flag←1	cc←0b0001 vxsnan_flag←1	cc←0b0001 vxsnan_flag←1

Explanation:

src1	The double-precision floating-point value in doubleword element 0 of VSR[XA].
src2	The double-precision floating-point value in doubleword element 0 of VSR[XB].
NZF	Nonzero finite number.
C(x,y)	The floating-point value x is compared to the floating-point value y, returning one of three 4-bit results. 0b1000 when x is greater than y 0b0100 when x is less than y 0b0010 when x is equal to y
cc	The 4-bit result compare code.

Table 7.60: Actions for xscmpudp - Part 1: Compare Unordered

VE	vxsnan_flag	Returned Results and Status Setting
–	0	FPCC←cc, CR[BF]←cc
0	1	FPCC←cc, CR[BF]←cc, fx(VXSNAN)
1	1	FPCC←cc, CR[BF]←cc, fx(VXSNAN), error()

Explanation:

–	The results do not depend on this condition.
cc	The 4-bit result as defined in Table 7.60.
error()	The system error handler is invoked for the trap-enabled exception if MSR.FE0 and MSR.FE1 are set to any mode other than the ignore-exception mode.
fx(x)	FPSCR.FX is set to 1 if FPSCR.x=0. FPSCR.x is set to 1.

Table 7.61: Actions for xscmpudp - Part 2: Result

VSX Scalar Compare Unordered Quad-Precision X-form

xscmpuqp BF,VRA,VRB

63	BF	//	VRA	VRB	644	/
0	6	9	11	16	21	31

```
if MSR.VSX=0 then VSX_Unavailable()

reset_xflags()

src1 ← bfp_CONVERT_FROM_BFP128(VSR[VRA+32])
src2 ← bfp_CONVERT_FROM_BFP128(VSR[VRB+32])

vxsnan_flag ← src1.class.SNaN | src2.class.SNaN

if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)

CR.bit[4×BF+32] ← FPSCR.FL ← src1 < src2
CR.bit[4×BF+33] ← FPSCR.FG ← src1 > src2
CR.bit[4×BF+34] ← FPSCR.FE ← src1 = src2
CR.bit[4×BF+35] ← FPSCR.FU ←
   src1.class.SNaN | src1.class.QNaN |
   src2.class.SNaN | src2.class.QNaN
```

Let src1 be the floating-point value in VSR[VRA+32] represented in quad-precision format.

Let src2 be the floating-point value in VSR[VRB+32] represented in quad-precision format.

src1 is compared to src2.

Zeros of same or opposite signs compare equal. Infinities of same signs compare equal.

Bit 0 of CR field BF and FL are set to indicate if src1 is less than src2.

Bit 1 of CR field BF and FG are set to indicate if src1 is greater than src2.

Bit 2 of CR field BF and FE are set to indicate if src1 is equal to src2.

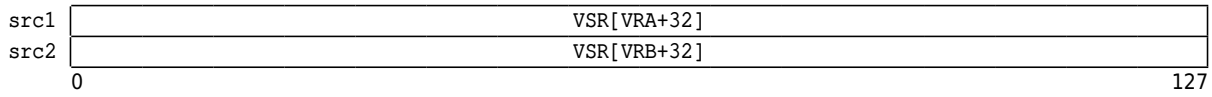
Bit 3 of CR field BF and FU are set to indicate unordered (i.e., src1 or src2 is a NaN).

If either of the operands is a Signaling NaN, an Invalid Operation exception occurs and VXSNAN is set to 1.

Special Registers Altered:

Register	*Field(s)*
CR	field BF
FPSCR	FPCC FX VXSNAN

VSR Data Layout for xscmpuqp



VSX Scalar Compare Equal Double-Precision XX3-form

xscmpeqdp XT,XA,XB

60	T	A	B	3	AX	BX	TX
0	6	11	16	21	29	30	31

```
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])

vxsnan_flag ← src1.class.SNaN | src2.class.SNaN

vex_flag ← FPSCR.VE & vxsnan_flag

if vxsnan_flag=1 SetFX(FPSCR.VXSNAN)

if vex_flag=0 then do
   if src1=src2 then
      VSR[32×TX+T].dword[0] ← 0xFFFF_FFFF_FFFF_FFFF
      VSR[32×TX+T].dword[1] ← 0x0000_0000_0000_0000
   end
   else do
      VSR[32×TX+T].dword[0] ← 0x0000_0000_0000_0000
      VSR[32×TX+T].dword[1] ← 0x0000_0000_0000_0000
   end
end
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

Let src1 be the double-precision floating-point value in doubleword 0 of VSR[XA].

Let src2 be the double-precision floating-point value in doubleword 0 of VSR[XB].

If src1 or src2 is a SNaN, an Invalid Operation exception occurs.

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.

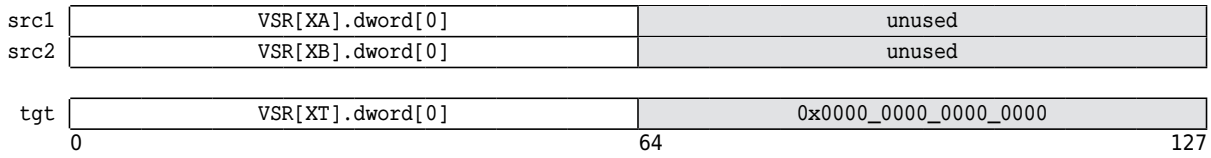
The contents of doubleword 1 of VSR[XT] are set to 0x0000_0000_0000_0000.

If a trap-enabled Invalid Operation occurs, VSR[XT] is not modified.

Special Registers Altered:

Register	*Field(s)*
FPSCR	FX VXSNAN

VSR Data Layout for xscmpeqdp



Programming Note

xscmpeqdp can be used to implement the C/C++/Java conditional operation, RESULT = (x=y) ? a:b.

```
xscmpeqdp   fEQ,fX,fY
xxsel       fRESULT,fA,fB,fEQ
```

VSX Scalar Compare Equal Quad-Precision X-form

xscmpeqqp VRT,VRA,VRB

63	VRT	VRA	VRB	68	/
0	6	11	16	21	31

```
if MSR.VSX=0 then VSX_Unavailable()

src1 ← bfp_CONVERT_FROM_BFP128(VSR[VRA+32])
src2 ← bfp_CONVERT_FROM_BFP128(VSR[VRB+32])

vxsnan_flag ← src1.class.SNaN | src2.class.SNaN

vex_flag ← FPSCR.VE & vxsnan_flag

if vxsnan_flag=1 SetFX(FPSCR.VXSNAN)

if vex_flag=0 then do
   if bfp_COMPARE_EQ(src1, src2)=1 then
      VSR[VRT+32] ←
         0xFFFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF
   else
      VSR[VRT+32] ←
         0x0000_0000_0000_0000_0000_0000_0000_0000
end
```

Let src1 be the quad-precision floating-point value in VSR[VRA+32].

Let src2 be the quad-precision floating-point value in VSR[VRB+32].

If src1 or src2 is a SNaN, an Invalid Operation exception occurs.

src1 is compared to src2.

A NaN compared to any value, including itself, compares false for the predicate, equal.

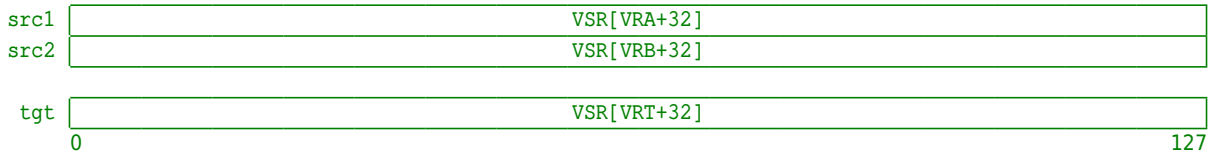
The contents of VSR[VRT+32] are set to all 1s if src1 is equal to src2, and are set to all 0s otherwise.

If a trap-enabled Invalid Operation occurs, VSR[VRT+32] is not modified.

Special Registers Altered:

Register	*Field(s)*
FPSCR	FX VXSNAN

VSR Data Layout for xscmpeqqp



Programming Note

xscmpeqqp can be used to implement the C/C++ conditional operation, RESULT = (x=y) ? a : b.

```
xscmpeqqp   vEQ,vX,vY
xxsel       vRESULT,vA,vB,vEQ
```

VSX Scalar Compare Greater Than or Equal Double-Precision XX3-form

xscmpgedp XT,XA,XB

60	T	A	B	19	AX	BX	TX
0	6	11	16	21	29	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

src1 ← bfp_CONVERT_FROM_BFP64(VSR[32×AX+A].dword[0])
src2 ← bfp_CONVERT_FROM_BFP64(VSR[32×BX+B].dword[0])

if src1.class.SNaN=1 | src2.class.SNaN=1 then do
   vxsnan_flag ← 0b1
   if FPSCR.VE=0 then vxvc_flag ← 0b1
end
else
   vxvc_flag ← src1.class.QNaN | src2.class.QNaN

vex_flag ← FPSCR.VE & (vxsnan_flag | vxvc_flag)

if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
if vxvc_flag=1   then SetFX(FPSCR.VXVC)

if vex_flag=0 then do
   if src1 >= src2 then
      VSR[32×TX+T].dword[0] ← 0xFFFF_FFFF_FFFF_FFFF
      VSR[32×TX+T].dword[1] ← 0x0000_0000_0000_0000
   end
   else do
      VSR[32×TX+T].dword[0] ← 0x0000_0000_0000_0000
      VSR[32×TX+T].dword[1] ← 0x0000_0000_0000_0000
   end
end
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

Let src1 be the double-precision floating-point value in doubleword 0 of VSR[XA].

Let src2 be the double-precision floating-point value in doubleword 0 of VSR[XB].

src1 is compared to src2.

A NaN compared to any value, including itself, compares false for the predicate, greater than or equal.

The contents of doubleword 0 of VSR[XT] are set to 0xFFFF_FFFF_FFFF_FFFF if src1 is greater than or equal to src2, and are set to 0x0000_0000_0000_0000 otherwise.

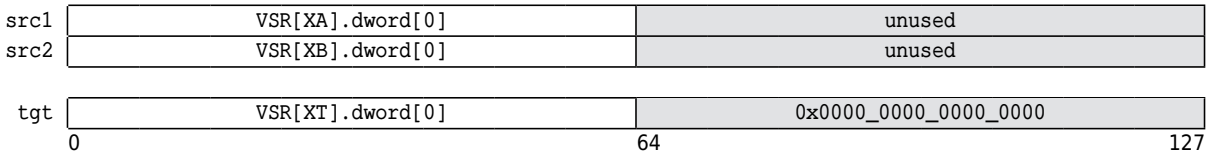
The contents of doubleword 1 of VSR[XT] are set to 0x0000_0000_0000_0000.

If a trap-enabled Invalid Operation occurs, VSR[XT] is not modified.

Special Registers Altered:

Register	*Field(s)*
FPSCR	FX VXSNAN VXVC

VSR Data Layout for xscmpgedp



Programming Note

xscmpgedp can be used to implement the C/C++/Java conditional operation, RESULT = (x>=y) ? a : b.

```
xscmpgedp   fGE,fX,fY
xxsel       fRESULT,fA,fB,fGE
```

xscmpgedp can also be used to implement the C/C++/Java conditional operation, RESULT = (x<=y) ? a : b.

```
xscmpgedp   fLE,fY,fX
xxsel       fRESULT,fA,fB,fLE
```

VSX Scalar Compare Greater Than or Equal Quad-Precision X-form

xscmpgeqp VRT,VRA,VRB

63	VRT	VRA	VRB	196	/
0	6	11	16	21	31

```
if MSR.VSX=0 then VSX_Unavailable()

src1 ← bfp_CONVERT_FROM_BFP128(VSR[VRA+32])
src2 ← bfp_CONVERT_FROM_BFP128(VSR[VRB+32])

if src1.class.SNaN=1 | src2.class.SNaN=1 then do
   vxsnan_flag ← 0b1
   if FPSCR.VE=0 then vxvc_flag ← 0b1
end
else
   vxvc_flag ← src1.class.QNaN | src2.class.QNaN

vex_flag ← FPSCR.VE & (vxsnan_flag | vxvc_flag)

if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
if vxvc_flag=1   then SetFX(FPSCR.VXVC)

if vex_flag=0 then do
   if bfp_COMPARE_GE(src1, src2)=1 then
      VSR[VRT+32] ←
         0xFFFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF
   else
      VSR[VRT+32] ←
         0x0000_0000_0000_0000_0000_0000_0000_0000
end
```

Let src1 be the quad-precision floating-point value in VSR[VRA+32].

Let src2 be the quad-precision floating-point value in VSR[VRB+32].

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 VSR[VRT+32] are set to all 1s if src1 is greater than or equal to src2, and are set to all 0s otherwise.

If a trap-enabled Invalid Operation occurs, VSR[VRT+32] is not modified.

Special Registers Altered:

Register	*Field(s)*
FPSCR	FX VXSNAN VXVC

VSR Data Layout for xscmpgeqp

Programming Note

xscmpgeqp can be used to implement the C/C++ conditional operation, RESULT = (x>=y) ? a : b.

```
xscmpgeqp   vGE,vX,vY
xxsel       vRESULT,vA,vB,vGE
```

xscmpgeqp can also be used to implement the C/C++ conditional operation, RESULT = (x<=y) ? a : b.

```
xscmpgeqp   vLE,vY,vX
xxsel       vRESULT,vA,vB,vLE
```

VSX Scalar Compare Greater Than Double-Precision XX3-form

xscmpgtdp XT,XA,XB

60	T	A	B	11	AX	BX	TX
0	6	11	16	21	29	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

src1 ← bfp_CONVERT_FROM_BFP64(VSR[32×AX+A].dword[0])
src2 ← bfp_CONVERT_FROM_BFP64(VSR[32×BX+B].dword[0])

if src1.class.SNaN=1 | src2.class.SNaN=1 then do
   vxsnan_flag ← 0b1
   if FPSCR.VE=0 then vxvc_flag ← 0b1
end
else
   vxvc_flag ← src1.class.QNaN | src2.class.QNaN

vex_flag ← FPSCR.VE & (vxsnan_flag | vxvc_flag)

if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
if vxvc_flag=1   then SetFX(FPSCR.VXVC)

if vex_flag=0 then do
   if src1 > src2 then
      VSR[32×TX+T].dword[0] ← 0xFFFF_FFFF_FFFF_FFFF
      VSR[32×TX+T].dword[1] ← 0x0000_0000_0000_0000
   end
   else do
      VSR[32×TX+T].dword[0] ← 0x0000_0000_0000_0000
      VSR[32×TX+T].dword[1] ← 0x0000_0000_0000_0000
   end
end
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

Let src1 be the double-precision floating-point value in doubleword 0 of VSR[XA].

Let src2 be the double-precision floating-point value in doubleword 0 of VSR[XB].

src1 is compared to src2.

A NaN compared to any value, including itself, compares false for the predicate, greater than.

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.

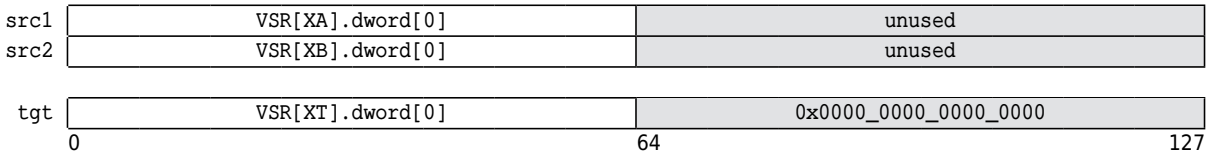
The contents of doubleword 1 of VSR[VRT] are set to 0x0000_0000_0000_0000.

If a trap-enabled Invalid Operation occurs, VSR[VRT+32] is not modified.

Special Registers Altered:

Register	*Field(s)*
FPSCR	FX VXSNAN VXVC

VSR Data Layout for xscmpgtdp



Programming Note

xscmpgtdp can be used to implement the C/C++/Java conditional operation, RESULT = (x>y) ? a : b.

```
xscmpgtdp   fGT,fX,fY
xxsel       fRESULT,fA,fB,fGT
```

xscmpgtdp can also be used to implement the C/C++/Java conditional operation, RESULT = (x<y) ? a : b.

```
xscmpgtdp   fLT,fY,fX
xxsel       fRESULT,fA,fB,fLT
```

VSX Scalar Compare Greater Than Quad-Precision X-form

xscmpgtqp VRT,VRA,VRB

63	VRT	VRA	VRB	228	/
0	6	11	16	21	31

```
if MSR.VSX=0 then VSX_Unavailable()

src1 ← bfp_CONVERT_FROM_BFP128(VSR[VRA+32])
src2 ← bfp_CONVERT_FROM_BFP128(VSR[VRB+32])

if src1.class.SNaN=1 | src2.class.SNaN=1 then do
   vxsnan_flag ← 0b1
   if FPSCR.VE=0 then vxvc_flag ← 0b1
end
else
   vxvc_flag ← src1.class.QNaN | src2.class.QNaN

vex_flag ← FPSCR.VE & (vxsnan_flag | vxvc_flag)

if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
if vxvc_flag=1   then SetFX(FPSCR.VXVC)

if vex_flag=0 then do
   if bfp_COMPARE_GT(src1, src2)=1 then
      VSR[VRT+32] ←
         0xFFFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF
   else
      VSR[VRT+32] ←
         0x0000_0000_0000_0000_0000_0000_0000_0000
end
```

Let src1 be the quad-precision floating-point value in VSR[VRA+32].

Let src2 be the quad-precision floating-point value in VSR[VRB+32].

src1 is compared to src2.

A NaN compared to any value, including itself, compares false for the predicate, greater than.

The contents of VSR[VRT+32] are set to all 1s if src1 is greater than src2, and are set to all 0s otherwise.

If a trap-enabled Invalid Operation occurs, VSR[VRT+32] is not modified.

Special Registers Altered:

Register	*Field(s)*
FPSCR	FX VXSNAN VXVC

VSR Data Layout for xscmpgtqp

Programming Note

xscmpgtqp can be used to implement the C/C++ conditional operation, RESULT = (x>y) ? a : b.

```
xscmpgtqp   vGT,vX,vY
xxsel       vRESULT,vA,vB,vGT
```

xscmpgtqp can also be used to implement the C/C++ conditional operation, RESULT = (x<y) ? a : b.

```
xscmpgtqp   vLT,vY,vX
xxsel       vRESULT,vA,vB,vLT
```

VSX Scalar Maximum Type-C Double-Precision XX3-form

xsmaxcdp XT,XA,XB

60	T	A	B	128	AX	BX	TX
0	6	11	16	21	29	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

reset_xflags()

src1   ← VSR[32×AX+A].dword[0]
src2   ← VSR[32×BX+B].dword[0]
result ← bfp64_MAXIMUM_TYPE_C(src1,src2)

if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)

vex_flag ← FPSCR.VE & vxsnan_flag

if vex_flag=0 then do
   VSR[32×TX+T].dword[0] ← result
   VSR[32×TX+T].dword[1] ← 0x0000_0000_0000_0000
end
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

Let src1 be the double-precision floating-point value in doubleword 0 of VSR[XA].

Let src2 be the double-precision floating-point value in doubleword 0 of VSR[XB].

If src1 or src2 is a SNaN, an Invalid Operation exception occurs.

If either src1 or src2 is a NaN, result is src2.

Otherwise, if src1 is greater than src2, result is src1.

Otherwise, result is src2.

The contents of doubleword 0 of VSR[XT] are set to the value result.

The contents of doubleword 1 of VSR[XT] are set to 0.

If a trap-enabled Invalid Operation occurs, VSR[XT] is not modified.

Special Registers Altered:

Register	*Field(s)*
FPSCR	FX VXSNAN

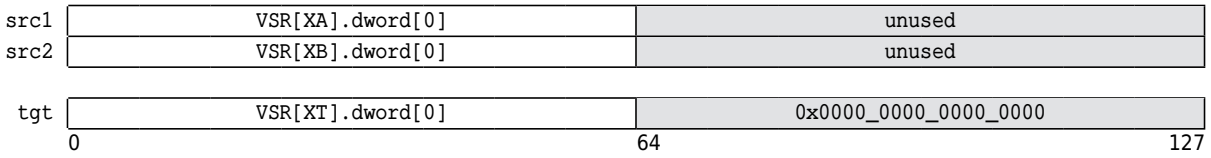
Programming Note

xsmaxcdp can be used to implement the C/C++/Java conditional operation (x>y)?x:y for single-precision and double-precision arguments.

Programming Note

Previous versions of the architecture allowed the contents of doubleword 1 of the result register to be undefined. However, all processors that support this instruction write 0s into doubleword 1 of the result register, as is required by this version of the architecture.

VSR Data Layout for xsmaxcdp



		src2							
		−Infinity	**−NZF**	**−Zero**	**+Zero**	**+NZF**	**+Infinity**	**QNaN**	**SNaN**
src1	**−Infinity**	T(src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(src2) fx(VXSNAN)
	−NZF	T(src1)	T(M(src1,src2))	T(src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(src2) fx(VXSNAN)
	−Zero	T(src1)	T(src1)	T(src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(src2) fx(VXSNAN)
	+Zero	T(src1)	T(src1)	T(src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(src2) fx(VXSNAN)
	+NZF	T(src1)	T(src1)	T(src1)	T(src1)	T(M(src1,src2))	T(src2)	T(src2)	T(src2) fx(VXSNAN)
	+Infinity	T(src1)	T(src1)	T(src1)	T(src1)	T(src1)	T(src2)	T(src2)	T(src2) fx(VXSNAN)
	QNaN	T(src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(src2) fx(VXSNAN)
	SNaN	T(src2) fx(VXSNAN)	T(src2) fx(VXSNAN)	T(src2) fx(VXSNAN)	T(src2) fx(VXSNAN)	T(src2) fx(VXSNAN)	T(src2) fx(VXSNAN)	T(src2) fx(VXSNAN)	T(src2) fx(VXSNAN)

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.
M(x,y)	Return the greater of floating-point value x and floating-point value y.
T(x)	The value x is placed in doubleword element 0 of `VSR[XT]` in double-precision format. The contents of doubleword element 1 of `VSR[XT]` are set to 0. 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 7.62: Actions for xsmaxcdp

VSX Scalar Maximum Type-C Quad-Precision X-form

xsmaxcqp VRT,VRA,VRB

63	VRT	VRA	VRB	676	/
0	6	11	16	21	31

```
if MSR.VSX=0 then VSX_Unavailable()

src1 ← bfp_CONVERT_FROM_BFP128(VSR[VRA+32])
src2 ← bfp_CONVERT_FROM_BFP128(VSR[VRB+32])

vxsnan_flag ← (src1.class.SNaN=1) |
              (src2.class.SNaN=1)

if (src1.class.SNaN=1) | (src1.class.QNaN=1) |
   (src2.class.SNaN=1) | (src2.class.QNaN=1) then
   result ← VSR[VRB+32]

else if bfp_COMPARE_GT(src1,src2) then
   result ← VSR[VRA+32]
else
   result ← VSR[VRB+32]

vex_flag ← FPSCR.VE & vxsnan_flag

if vxsnan_flag=1 then SetFX(VXSNAN)

if vex_flag=0 then
   VSR[VRT+32] ← result
```

Let src1 be the quad-precision floating-point value in VSR[VRA+32].

Let src2 be the quad-precision floating-point value in VSR[VRB+32].

If src1 or src2 is a SNaN, an Invalid Operation exception occurs.

If either src1 or src2 is a NaN, result is src2.

Otherwise, if src1 is greater than src2, result is src1.

Otherwise, result is src2.

The contents of VSR[VRT+32] are set to the value result.

If a trap-enabled Invalid Operation occurs, VSR[VRT+32] is not modified.

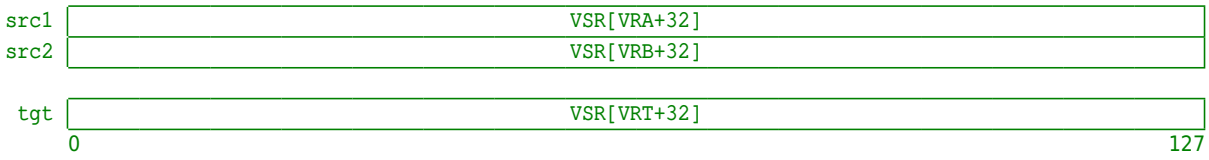
Special Registers Altered:

Register	*Field(s)*
FPSCR	FX VXSNAN

Programming Note

xsmaxcqp can be used to implement the C/C++ conditional operation (x>y)?x:y for quad-precision arguments.

VSR Data Layout for xsmaxcqp



src1 \ src2	−Infinity	−NZF	−Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
−Infinity	T(src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(src2) fx(VXSNAN)
−NZF	T(src1)	T(M(src1,src2))	T(src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(src2) fx(VXSNAN)
−Zero	T(src1)	T(src1)	T(src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(src2) fx(VXSNAN)
+Zero	T(src1)	T(src1)	T(src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(src2) fx(VXSNAN)
+NZF	T(src1)	T(src1)	T(src1)	T(src1)	T(M(src1,src2))	T(src2)	T(src2)	T(src2) fx(VXSNAN)
+Infinity	T(src1)	T(src1)	T(src1)	T(src1)	T(src1)	T(src2)	T(src2)	T(src2) fx(VXSNAN)
QNaN	T(src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(src2) fx(VXSNAN)
SNaN	T(src2) fx(VXSNAN)	T(src2) fx(VXSNAN)	T(src2) fx(VXSNAN)	T(src2) fx(VXSNAN)	T(src2) fx(VXSNAN)	T(src2) fx(VXSNAN)	T(src2) fx(VXSNAN)	T(src2) fx(VXSNAN)

Explanation:

src1	The quad-precision floating-point value in `VSR[VRA+32]`.
src2	The quad-precision floating-point value in `VSR[VRB+32]`.
NZF	Nonzero finite number.
M(x,y)	Return the greater of floating-point value x and floating-point value y.
T(x)	The value x is placed in `VSR[XT]` in quad-precision format. FPRF, FR and FI are not modified.
fx(x)	If x is equal to 0, FX is set to 1. x is set to 1.
VXSNAN	Floating-Point Invalid Operation Exception (SNaN) status flag, $FPSCR_{VXSNAN}$. If VE=1, update of `VSR[VRT+32]` is suppressed.

Table 7.63: Actions for xsmaxcqp

VSX Scalar Maximum Double-Precision XX3-form

xsmaxdp XT,XA,XB

60	T	A	B	160	AX	BX	TX
0	6	11	16	21	29	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

reset_xflags()

src1   ← VSR[32×AX+A].dword[0]
src2   ← VSR[32×BX+B].dword[0]
result ← bfp64_MAXIMUM(src1,src2)

if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)

vex_flag ← FPSCR.VE & vxsnan_flag

if vex_flag=0 then do
   VSR[32×TX+T].dword[0] ← result
   VSR[32×TX+T].dword[1] ← 0x0000_0000_0000_0000
end
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

Let src1 be the double-precision floating-point value in doubleword element 0 of VSR[XA].

Let src2 be the double-precision floating-point value in doubleword element 0 of VSR[XB].

If src1 is greater than src2, src1 is placed into doubleword element 0 of VSR[XT]. Otherwise, src2 is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are set to 0.

The maximum of +0 and −0 is +0. The maximum of a QNaN and any value is that value. The maximum of any value and an SNaN is that SNaN converted to a QNaN.

FPRF, FR and FI are not modified.

If a trap-enabled invalid operation exception occurs, VSR[XT] is not modified.

See Table 7.64.

Special Registers Altered:

Register	*Field(s)*
FPSCR	FX VXSNAN

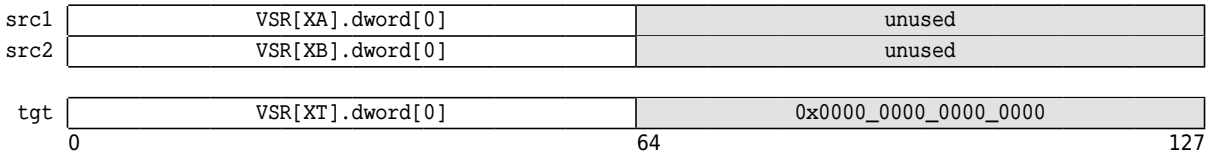
Programming Note

This instruction can be used to operate on single-precision source operands.

Programming Note

Previous versions of the architecture allowed the contents of doubleword 1 of the result register to be undefined. However, all processors that support this instruction write 0s into doubleword 1 of the result register, as is required by this version of the architecture.

VSR Data Layout for xsmaxdp



src1 \ src2	−Infinity	−NZF	−Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
−Infinity	T(src1)	T(src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(src1)	T(Q(src2)) fx(VXSNAN)
−NZF	T(src1)	T(M(src1,src2))	T(src2)	T(src2)	T(src2)	T(src2)	T(src1)	T(Q(src2)) fx(VXSNAN)
−Zero	T(src1)	T(src1)	T(src1)	T(src2)	T(src2)	T(src2)	T(src1)	T(Q(src2)) fx(VXSNAN)
+Zero	T(src1)	T(src1)	T(src1)	T(src1)	T(src2)	T(src2)	T(src1)	T(Q(src2)) fx(VXSNAN)
+NZF	T(src1)	T(src1)	T(src1)	T(src1)	T(M(src1,src2))	T(src2)	T(src1)	T(Q(src2)) fx(VXSNAN)
+Infinity	T(src1)	T(src1)	T(src1)	T(src1)	T(src1)	T(src1)	T(src1)	T(Q(src2)) fx(VXSNAN)
QNaN	T(src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(src1)	T(src1) fx(VXSNAN)
SNaN	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)

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.
Q(x)	Return a QNaN with the payload of x.
M(x,y)	Return the greater of floating-point value x and floating-point value y.
T(x)	The value x is placed in doubleword element 0 of VSR[XT] in double-precision format. The contents of doubleword element 1 of VSR[XT] are set to 0. 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 7.64: Actions for xsmaxdp

VSX Scalar Maximum Type-J Double-Precision XX3-form

xsmaxjdp XT,XA,XB

60	T	A	B	144	AX	BX	TX
0	6	11	16	21	29	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

reset_xflags()

src1   ← VSR[32×AX+A].dword[0]
src2   ← VSR[32×BX+B].dword[0]
result ← bfp64_MAXIMUM_TYPE_J(src1,src2)

if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)

vex_flag ← FPSCR.VE & vxsnan_flag

if vex_flag=0 then do
   VSR[32×TX+T].dword[0] ← result
   VSR[32×TX+T].dword[1] ← 0x0000_0000_0000_0000
end
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

Let src1 be the double-precision floating-point value in doubleword 0 of VSR[XA].

Let src2 be the double-precision floating-point value in doubleword 0 of VSR[XB].

If src1 or src2 is a SNaN, an Invalid Operation exception occurs.

If src1 is a NaN, result is src1.

Otherwise, if src2 is a NaN, result is src2.

Otherwise, if src1 is a Zero and src2 is a Zero and either src1 or src2 is a +Zero, the result is +Zero.

Otherwise, if src1 is a −Zero and src2 is a −Zero, the result is −Zero.

Otherwise, if src1 is greater than src2, result is src1.

Otherwise, result is src2.

The contents of doubleword 0 of VSR[XT] are set to the value result.

The contents of doubleword 1 of VSR[XT] are set to 0.

If a trap-enabled Invalid Operation occurs, VSR[XT] is not modified.

Special Registers Altered:

Register	*Field(s)*
FPSCR	FX VXSNAN

Programming Note

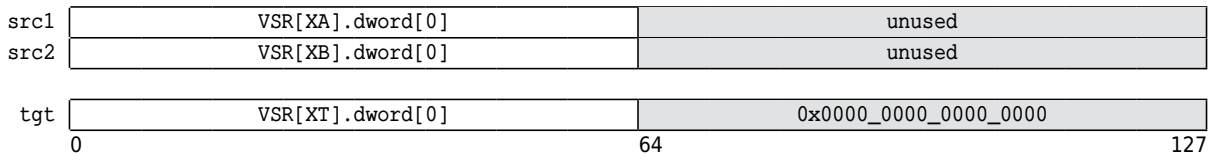
xsmaxjdp can be used to implement the Java max() function for single-precision and double-precision arguments.

Despite Java not recognizing the concept of exception status, VXSNAN is set to 1 if either operand is a SNaN.

Programming Note

Previous versions of the architecture allowed the contents of doubleword 1 of the result register to be undefined. However, all processors that support this instruction write 0s into doubleword 1 of the result register, as is required by this version of the architecture.

VSR Data Layout for xsmaxjdp



		src2							
		−Infinity	**−NZF**	**−Zero**	**+Zero**	**+NZF**	**+Infinity**	**QNaN**	**SNaN**
src1	**−Infinity**	T(−INF)	T(src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(src2) fx(VXSNAN)
	−NZF	T(src1)	T(M(src1,src2))	T(src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(src2) fx(VXSNAN)
	−Zero	T(src1)	T(src1)	T(−Zero)	T(+Zero)	T(src2)	T(src2)	T(src2)	T(src2) fx(VXSNAN)
	+Zero	T(src1)	T(src1)	T(+Zero)	T(+Zero)	T(src2)	T(src2)	T(src2)	T(src2) fx(VXSNAN)
	+NZF	T(src1)	T(src1)	T(src1)	T(src1)	T(M(src1,src2))	T(src2)	T(src2)	T(src2) fx(VXSNAN)
	+Infinity	T(src1)	T(src1)	T(src1)	T(src1)	T(src1)	T(+INF)	T(src2)	T(src2) fx(VXSNAN)
	QNaN	T(src1)	T(src1)	T(src1)	T(src1)	T(src1)	T(src1)	T(src1)	T(src1) fx(VXSNAN)
	SNaN	T(src1) fx(VXSNAN)	T(src1) fx(VXSNAN)	T(src1) fx(VXSNAN)	T(src1) fx(VXSNAN)	T(src1) fx(VXSNAN)	T(src1) fx(VXSNAN)	T(src1) fx(VXSNAN)	T(src1) fx(VXSNAN)

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.
M(x,y)	Return the greater of floating-point value x and floating-point value y.
T(x)	The value x is placed in doubleword element 0 of `VSR[XT]` in double-precision format. The contents of doubleword element 1 of `VSR[XT]` are set to 0. 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 7.65: Actions for xsmaxjdp

VSX Scalar Minimum Type-C Double-Precision XX3-form

xsmincdp XT,XA,XB

60	T	A	B	136	AX	BX	TX
0	6	11	16	21	29	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

reset_xflags()

src1   ← VSR[32×AX+A].dword[0]
src2   ← VSR[32×BX+B].dword[0]
result ← bfp64_MINIMUM_TYPE_C(src1,src2)

if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)

vex_flag ← FPSCR.VE & vxsnan_flag

if vex_flag=0 then do
   VSR[32×TX+T].dword[0] ← result
   VSR[32×TX+T].dword[1] ← 0x0000_0000_0000_0000
end
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

Let src1 be the double-precision floating-point value in doubleword 0 of VSR[XA].

Let src2 be the double-precision floating-point value in doubleword 0 of VSR[XB].

If src1 or src2 is a SNaN, an Invalid Operation exception occurs.

If either src1 or src2 is a NaN, result is src2.

Otherwise, if src1 is less than src2, result is src1.

Otherwise, result is src2.

The contents of doubleword 0 of VSR[XT] are set to the value result.

The contents of doubleword 1 of VSR[XT] are set to 0.

If a trap-enabled Invalid Operation occurs, VSR[XT] is not modified.

Special Registers Altered:

Register	*Field(s)*
FPSCR	FX VXSNAN

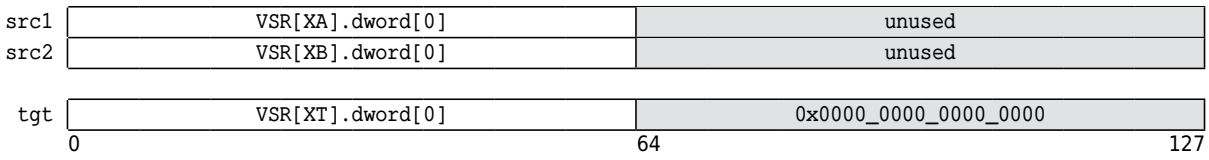
Programming Note

xsmincdp can be used to implement the C/C++/Java conditional operator (x<y)?x:y for single-precision and double-precision arguments.

Programming Note

Previous versions of the architecture allowed the contents of doubleword 1 of the result register to be undefined. However, all processors that support this instruction write 0s into doubleword 1 of the result register, as is required by this version of the architecture.

VSR Data Layout for xsmincdp



		src2							
		−Infinity	**−NZF**	**−Zero**	**+Zero**	**+NZF**	**+Infinity**	**QNaN**	**SNaN**
src1	**−Infinity**	T(src2)	T(src1)	T(src1)	T(src1)	T(src1)	T(src1)	T(src2)	T(src2) fx(VXSNAN)
	−NZF	T(src2)	T(M(src1,src2))	T(src1)	T(src1)	T(src1)	T(src1)	T(src2)	T(src2) fx(VXSNAN)
	−Zero	T(src2)	T(src2)	T(src2)	T(src2)	T(src1)	T(src1)	T(src2)	T(src2) fx(VXSNAN)
	+Zero	T(src2)	T(src2)	T(src2)	T(src2)	T(src1)	T(src1)	T(src2)	T(src2) fx(VXSNAN)
	+NZF	T(src2)	T(src2)	T(src2)	T(src2)	T(M(src1,src2))	T(src1)	T(src2)	T(src2) fx(VXSNAN)
	+Infinity	T(src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(src2) fx(VXSNAN)
	QNaN	T(src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(src2) fx(VXSNAN)
	SNaN	T(src2) fx(VXSNAN)	T(src2) fx(VXSNAN)	T(src2) fx(VXSNAN)	T(src2) fx(VXSNAN)	T(src2) fx(VXSNAN)	T(src2) fx(VXSNAN)	T(src2) fx(VXSNAN)	T(src2) fx(VXSNAN)

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.
M(x,y)	Return the lesser of floating-point value x and floating-point value y.
T(x)	The value x is placed in doubleword element 0 of `VSR[XT]` in double-precision format. The contents of doubleword element 1 of `VSR[XT]` are set to 0. 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 7.66: Actions for xsmincdp

VSX Scalar Minimum Type-C Quad-Precision X-form

xsmincqp VRT,VRA,VRB

63	VRT	VRA	VRB	740	/
0	6	11	16	21	31

```
if MSR.VSX=0 then VSX_Unavailable()

src1 ← bfp_CONVERT_FROM_BFP128(VSR[VRA+32])
src2 ← bfp_CONVERT_FROM_BFP128(VSR[VRB+32])

vxsnan_flag ← (src1.class.SNaN=1) |
              (src2.class.SNaN=1)

if (src1.class.SNaN=1) | (src1.class.QNaN=1) |
   (src2.class.SNaN=1) | (src2.class.QNaN=1) then
   result ← VSR[VRB+32]

else if bfp_COMPARE_LT(src1,src2) then
   result ← VSR[VRA+32]
else
   result ← VSR[VRB+32]

vex_flag ← FPSCR.VE & vxsnan_flag

if vxsnan_flag=1 then SetFX(VXSNAN)

if vex_flag=0 then
   VSR[VRT+32] ← result
```

Let `src1` be the quad-precision floating-point value in `VSR[VRA+32]`.

Let `src2` be the quad-precision floating-point value in `VSR[VRB+32]`.

If `src1` or `src2` is a SNaN, an Invalid Operation exception occurs.

If either `src1` or `src2` is a NaN, `result` is `src2`.

Otherwise, if `src1` is less than `src2`, `result` is `src1`.

Otherwise, `result` is `src2`.

The contents of `VSR[VRT+32]` are set to the value `result`.

If a trap-enabled Invalid Operation occurs, `VSR[VRT+32]` is not modified.

Special Registers Altered:

Register	*Field(s)*
FPSCR	FX VXSNAN

Programming Note

xsmincqp can be used to implement the C/C++ conditional operator `(x<y)?x:y` for quad-precision arguments.

src1 \ src2	−Infinity	−NZF	−Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
−Infinity	T(src2)	T(src1)	T(src1)	T(src1)	T(src1)	T(src1)	T(src2)	T(src2) fx(VXSNAN)
−NZF	T(src2)	T(M(src1,src2))	T(src1)	T(src1)	T(src1)	T(src1)	T(src2)	T(src2) fx(VXSNAN)
−Zero	T(src2)	T(src2)	T(src2)	T(src2)	T(src1)	T(src1)	T(src2)	T(src2) fx(VXSNAN)
+Zero	T(src2)	T(src2)	T(src2)	T(src2)	T(src1)	T(src1)	T(src2)	T(src2) fx(VXSNAN)
+NZF	T(src2)	T(src2)	T(src2)	T(src2)	T(M(src1,src2))	T(src1)	T(src2)	T(src2) fx(VXSNAN)
+Infinity	T(src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(src2) fx(VXSNAN)
QNaN	T(src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(src2) fx(VXSNAN)
SNaN	T(src2) fx(VXSNAN)	T(src2) fx(VXSNAN)	T(src2) fx(VXSNAN)	T(src2) fx(VXSNAN)	T(src2) fx(VXSNAN)	T(src2) fx(VXSNAN)	T(src2) fx(VXSNAN)	T(src2) fx(VXSNAN)

Explanation:

src1	The quad-precision floating-point value in `VSR[VRA+32]`.
src2	The quad-precision floating-point value in `VSR[VRB+32]`.
NZF	Nonzero finite number.
M(x,y)	Return the lesser of floating-point value x and floating-point value y.
T(x)	The value x is placed in `VSR[VRT+32]` in quad-precision format. FPRF, FR and FI are not modified.
fx(x)	If x is equal to 0, FX is set to 1. x is set to 1.
VXSNAN	Floating-Point Invalid Operation Exception (SNaN) status flag, $FPSCR_{VXSNAN}$. If VE=1, update of `VSR[VRT+32]` is suppressed.

Table 7.67: Actions for xsmincqp

VSX Scalar Minimum Double-Precision XX3-form

xsmindp XT,XA,XB

60	T	A	B	168	AX	BX	TX
0	6	11	16	21	29	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

reset_xflags()

src1   ← VSR[32×AX+A].dword[0]
src2   ← VSR[32×BX+B].dword[0]
result ← bfp64_MINIMUM(src1,src2)

if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)

vex_flag ← FPSCR.VE & vxsnan_flag

if vex_flag=0 then do
   VSR[32×TX+T].dword[0] ← result
   VSR[32×TX+T].dword[1] ← 0x0000_0000_0000_0000
end
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

Let src1 be the double-precision floating-point value in doubleword element 0 of VSR[XA].

Let src2 be the double-precision floating-point value in doubleword element 0 of VSR[XB].

If src1 is less than src2, src1 is placed into doubleword element 0 of VSR[XT] in double-precision format. Otherwise, src2 is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are set to 0.

The minimum of +0 and –0 is –0. The minimum of a QNaN and any value is that value. The minimum of any value and an SNaN is that SNaN converted to a QNaN.

FPRF, FR and FI are not modified.

If a trap-enabled invalid operation exception occurs, VSR[XT] is not modified.

See Table 7.68.

Special Registers Altered:

Register	*Field(s)*
FPSCR	FX VXSNAN

Programming Note

This instruction can be used to operate on single-precision source operands.

Programming Note

Previous versions of the architecture allowed the contents of doubleword 1 of the result register to be undefined. However, all processors that support this instruction write 0s into doubleword 1 of the result register, as is required by this version of the architecture.

VSR Data Layout for xsmindp

src1	VSR[XA].dword[0]	unused
src2	VSR[XB].dword[0]	unused

tgt	VSR[XT].dword[0]	0x0000_0000_0000_0000

0 64 127

src1 \ src2	−Infinity	−NZF	−Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
−Infinity	T(src1)	T(src1)	T(src1)	T(src1)	T(src1)	T(src1)	T(src1)	T(Q(src2)) fx(VXSNAN)
−NZF	T(src2)	T(M(src1,src2))	T(src1)	T(src1)	T(src1)	T(src1)	T(src1)	T(Q(src2)) fx(VXSNAN)
−Zero	T(src2)	T(src2)	T(src1)	T(src1)	T(src1)	T(src1)	T(src1)	T(Q(src2)) fx(VXSNAN)
+Zero	T(src2)	T(src2)	T(src2)	T(src1)	T(src1)	T(src1)	T(src1)	T(Q(src2)) fx(VXSNAN)
+NZF	T(src2)	T(src2)	T(src2)	T(src2)	T(M(src1,src2))	T(src1)	T(src1)	T(Q(src2)) fx(VXSNAN)
+Infinity	T(src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(src1)	T(src1)	T(Q(src2)) fx(VXSNAN)
QNaN	T(src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(src1)	T(src1) fx(VXSNAN)
SNaN	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)

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.
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 0 of VSR[XT] in double-precision format. The contents of doubleword element 1 of VSR[XT] are set to 0. FPRF, FR and FI are not modified.
fx(x)	If x is equal to 0, FX is set to 1. x is set to 1.
VXSNAN	Floating-Point Invalid Operation Exception (SNaN) status flag, $FPSCR_{VXSNAN}$. If VE=1, update of VSR[XT] is suppressed.

Table 7.68: Actions for xsmindp

VSX Scalar Minimum Type-J Double-Precision XX3-form

xsminjdp XT,XA,XB

60	T	A	B	152	AX	BX	TX
0	6	11	16	21	29	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

reset_xflags()

src1   ← VSR[32×AX+A].dword[0]
src2   ← VSR[32×BX+B].dword[0]
result ← bfp64_MINIMUM_TYPE_J(src1,src2)

if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)

vex_flag ← FPSCR.VE & vxsnan_flag

if vex_flag=0 then do
   VSR[32×TX+T].dword[0] ← result
   VSR[32×TX+T].dword[1] ← 0x0000_0000_0000_0000
end
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

Let src1 be the double-precision floating-point value in doubleword 0 of VSR[XA].

Let src2 be the double-precision floating-point value in doubleword 0 of VSR[XB].

If src1 or src2 is a SNaN, an Invalid Operation exception occurs.

If src1 is a NaN, result is src1.

Otherwise, if src2 is a NaN, result is src2.

Otherwise, if src1 is a Zero and src2 is a Zero and either src1 or src2 is a −Zero, the result is −Zero.

Otherwise, if src1 is a +Zero and src2 is a +Zero, the result is +Zero.

Otherwise, if src1 is less than src2, result is src1.

Otherwise, result is src2.

The contents of doubleword 0 of VSR[XT] are set to the value result.

The contents of doubleword 1 of VSR[XT] are set to 0.

If a trap-enabled Invalid Operation occurs, VSR[XT] is not modified.

Special Registers Altered:

Register	*Field(s)*
FPSCR	FX VXSNAN

Programming Note

xsminjdp can be used to implement the Java min() function for single-precision and double-precision arguments.

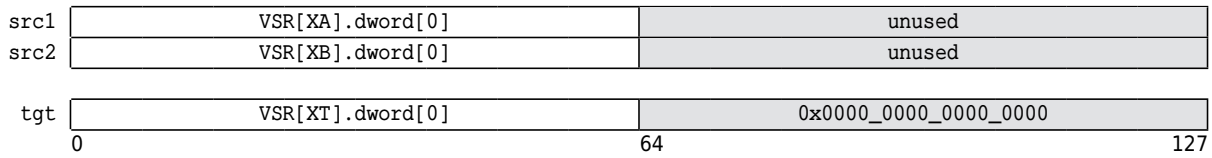
Note that Java only recognizes the concept of a NaN, but does not distinguish any difference between different NaN encodings, including between a QNaN and a SNaN. As a result, a SNaN operand is propagated as a SNaN (i.e., not converted to a QNaN).

Despite Java not recognizing the concept of exception status, VXSNAN is set to 1 if either operand is a SNaN.

Programming Note

Previous versions of the architecture allowed the contents of doubleword 1 of the result register to be undefined. However, all processors that support this instruction write 0s into doubleword 1 of the result register, as is required by this version of the architecture.

VSR Data Layout for xsminjdp



		src2							
		−Infinity	**−NZF**	**−Zero**	**+Zero**	**+NZF**	**+Infinity**	**QNaN**	**SNaN**
src1	**−Infinity**	T(−INF)	T(src1)	T(src1)	T(src1)	T(src1)	T(src1)	T(src2)	T(src2) fx(VXSNAN)
	−NZF	T(src2)	T(M(src1,src2))	T(src1)	T(src1)	T(src1)	T(src1)	T(src2)	T(src2) fx(VXSNAN)
	−Zero	T(src2)	T(src2)	T(−Zero)	T(−Zero)	T(src1)	T(src1)	T(src2)	T(src2) fx(VXSNAN)
	+Zero	T(src2)	T(src2)	T(−Zero)	T(+Zero)	T(src1)	T(src1)	T(src2)	T(src2) fx(VXSNAN)
	+NZF	T(src2)	T(src2)	T(src2)	T(src2)	T(M(src1,src2))	T(src1)	T(src2)	T(src2) fx(VXSNAN)
	+Infinity	T(src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(+INF)	T(src2)	T(src2) fx(VXSNAN)
	QNaN	T(src1)	T(src1)	T(src1)	T(src1)	T(src1)	T(src1)	T(src1)	T(src1) fx(VXSNAN)
	SNaN	T(src1) fx(VXSNAN)	T(src1) fx(VXSNAN)	T(src1) fx(VXSNAN)	T(src1) fx(VXSNAN)	T(src1) fx(VXSNAN)	T(src1) fx(VXSNAN)	T(src1) fx(VXSNAN)	T(src1) fx(VXSNAN)

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.
M(x,y)	Return the lesser of floating-point value x and floating-point value y.
T(x)	The value x is placed in doubleword element 0 of `VSR[XT]` in double-precision format. The contents of doubleword element 1 of `VSR[XT]` are set to 0. 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 7.69: Actions for xsminjdp

7.6.2.4.2 VSX Vector BFP Compare Instructions

VSX Vector Compare Equal To Double-Precision XX3-form

xvcmpeqdp	XT,XA,XB	(Rc=0)
xvcmpeqdp.	XT,XA,XB	(Rc=1)

60	T	A	B	Rc	99	AX	BX	TX
0	6	11	16	21	22	29	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

ex_flag   ← 0b0
all_false ← 0b1
all_true  ← 0b1

do i = 0 to 1
   reset_xflags()

   src1 ← bfp_CONVERT_FROM_BFP64(VSR[32×AX+A].dword[i])
   src2 ← bfp_CONVERT_FROM_BFP64(VSR[32×BX+B].dword[i])

   vxsnan_flag ← IsSNaN(src1) | IsSNaN(src2)

   if src1 = src2 then do
      vresult.dword[i] ← 0xFFFF_FFFF_FFFF_FFFF
      all_false        ← 0b0
   end
   else do
      vresult.dword[i] ← 0x0000_0000_0000_0000
      all_true         ← 0b0
   end

   if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)

   ex_flag ← ex_flag | (FPSCR.VE & vxsnan_flag)
end

if ex_flag=0 then VSR[32×TX+T] ← vresult

if Rc=1 then do
   if vex_flag=0 then
      CR[6] ← all_true || 0b0 || all_false || 0b0
   else
      CR[6] ← 0bUUUU
end
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

For each integer value i from 0 to 1, do the following.

Let src1 be the double-precision floating-point operand in doubleword element i of VSR[XA].

Let src2 be the double-precision floating-point operand in doubleword element i of VSR[XB].

src1 is compared to src2.

The contents of doubleword element i of VSR[XT] are set to all 1s if src1 is equal to src2, and is set to all 0s otherwise.

A NaN input causes the comparison to return false for that element.

Two zero inputs of same or different signs return true for that element.

Two infinity inputs of same signs return true for that element.

If Rc=1, CR Field 6 is set as follows.

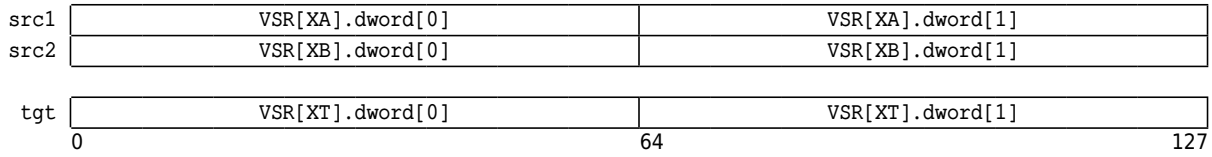
- Bit 0 is set to indicate all vector elements compared true.
- Bit 1 is set to 0.
- Bit 2 is set to indicate all vector elements compared false.
- Bit 3 is set to 0.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT] and the contents of CR field 6 are undefined if Rc is equal to 1.

Special Registers Altered:

Register	*Field(s)*	
CR	CR6	(if Rc=1)
FPSCR	FX VXSNAN	

VSR Data Layout for xvcmpeqdp[.]



VSX Vector Compare Equal To Single-Precision XX3-form

xvcmpeqsp	XT,XA,XB	(Rc=0)
xvcmpeqsp.	XT,XA,XB	(Rc=1)

60	T	A	B	Rc	67	AX	BX	TX
0	6	11	16	21	22	29	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

ex_flag   ← 0b0
all_false ← 0b1
all_true  ← 0b1

do i = 0 to 3
   reset_xflags()

   src1 ← bfp_CONVERT_FROM_BFP32(VSR[32×AX+A].word[i])
   src2 ← bfp_CONVERT_FROM_BFP32(VSR[32×BX+B].word[i])

   vxsnan_flag ← IsSNaN(src1) | IsSNaN(src2)

   if src1 = src2 then do
      vresult.word[i] ← 0xFFFF_FFFF
      all_false       ← 0b0
   end
   else do
      vresult.word[i] ← 0x0000_0000
      all_true        ← 0b0
   end

   if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)

   ex_flag ← ex_flag | (FPSCR.VE & vxsnan_flag)
end

if ex_flag=0 then VSR[32×TX+T] ← vresult

if Rc=1 then do
   if vex_flag=0 then
      CR.field[6] ← all_true || 0b0 || all_false ||
0b0
   else
      CR.field[6] ← 0bUUUU
end
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

For each integer value i from 0 to 3, do the following.

Let src1 be the single-precision floating-point operand in word element i of VSR[XA].

Let src2 be the single-precision floating-point operand in word element i of VSR[XB].

src1 is compared to src2.

The contents of word element i of VSR[XT] are set to all 1s if src1 is equal to src2, and is set to all 0s otherwise.

A NaN input causes the comparison to return false for that element.

Two zero inputs of same or different signs return true for that element.

Two infinity inputs of same signs return true for that element.

If Rc=1, CR Field 6 is set as follows.

- Bit 0 is set to indicate all vector elements compared true.
- Bit 1 is set to 0.
- Bit 2 is set to indicate all vector elements compared false.
- Bit 3 is set to 0.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT] and the contents of CR field 6 are undefined if Rc is equal to 1.

Special Registers Altered:

Register	*Field(s)*	
CR	CR6	(if Rc=1)
FPSCR	FX VXSNAN	

VSR Data Layout for xvcmpeqsp[.]

src1	VSR[XA].word[0]	VSR[XA].word[1]	VSR[XA].word[2]	VSR[XA].word[3]
src2	VSR[XB].word[0]	VSR[XB].word[1]	VSR[XB].word[2]	VSR[XB].word[3]
tgt	VSR[XT].word[0]	VSR[XT].word[1]	VSR[XT].word[2]	VSR[XT].word[3]
	0	32	64	96 127

VSX Vector Compare Greater Than or Equal To Double-Precision XX3-form

xvcmpgedp	XT,XA,XB	(Rc=0)
xvcmpgedp.	XT,XA,XB	(Rc=1)

60	T	A	B	Rc	115	AX	BX	TX
0	6	11	16	21	22	29	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

ex_flag   ← 0b0
all_false ← 0b1
all_true  ← 0b1

do i = 0 to 1
   reset_xflags()

   src1 ← bfp_CONVERT_FROM_BFP64(VSR[32×AX+A].dword[i])
   src2 ← bfp_CONVERT_FROM_BFP64(VSR[32×BX+B].dword[i])

   if src1.class.SNaN | src2.class.SNaN then do
      vxsnan_flag ← 0b1
      if FPSCR.VE=0 then vxvc_flag ← 0b1
   end
   else vxvc_flag ← IsQNaN(src1) | IsQNaN(src2)

   if src1 >= src2 then do
      vresult.dword[i] ← 0xFFFF_FFFF_FFFF_FFFF
      all_false        ← 0b0
   end
   else do
      vresult.dword[i] ← 0x0000_0000_0000_0000
      all_true         ← 0b0
   end

   if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
   if vxvc_flag=1   then SetFX(FPSCR.VXVC)

   ex_flag ← ex_flag | (FPSCR.VE & vxsnan_flag)
                     | (FPSCR.VE & vxvc_flag)
end

if ex_flag=0 then VSR[32×TX+T] ← vresult

if Rc=1 then do
   if vex_flag=0 then
      CR.field[6] ← all_true || 0b0 || all_false ||
0b0
   else
      CR.field[6] ← 0bUUUU
end
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

For each integer value i from 0 to 1, do the following.

Let src1 be the double-precision floating-point operand in doubleword element i of VSR[XA].

Let src2 be the double-precision floating-point operand in doubleword element i of VSR[XB].

src1 is compared to src2.

The contents of doubleword element i of VSR[XT] are set to all 1s if src1 is greater than or equal to the double-precision floating-point operand in doubleword element i of src2, and is set to all 0s otherwise.

A NaN input causes the comparison to return false for that element.

Two zero inputs of same or different signs return true for that element.

Two infinity inputs of same signs return true for that element.

If Rc=1, CR Field 6 is set as follows.

- Bit 0 is set to indicate all vector elements compared true.
- Bit 1 is set to 0.
- Bit 2 is set to indicate all vector elements compared false.
- Bit 3 is set to 0.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT] and the contents of CR field 6 are undefined if Rc is equal to 1.

Special Registers Altered:

Register	*Field(s)*	
CR	CR6	(if Rc=1)
FPSCR	FX VXSNAN VXVC	

VSR Data Layout for xvcmpgedp[.]

src1	VSR[XA].dword[0]	VSR[XA].dword[1]
src2	VSR[XB].dword[0]	VSR[XB].dword[1]

tgt	VSR[XT].dword[0]	VSR[XT].dword[1]

0 64 127

VSX Vector Compare Greater Than or Equal To Single-Precision XX3-form

xvcmpgesp	XT,XA,XB	(Rc=0)
xvcmpgesp.	XT,XA,XB	(Rc=1)

60	T	A	B	Rc	83	AX	BX	TX
0	6	11	16	21	22	29	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

ex_flag   ← 0b0
all_false ← 0b1
all_true  ← 0b1

do i=0 to 3
   reset_xflags()

   src1 ← bfp_CONVERT_FROM_BFP32(VSR[32×AX+A].word[i])
   src2 ← bfp_CONVERT_FROM_BFP32(VSR[32×BX+B].word[i])

   if src1.class.SNaN | src2.class.SNaN then do
      vxsnan_flag ← 0b1
      if FPSCR.VE=0 then vxvc_flag ← 0b1
   end
   else vxvc_flag ← IsQNaN(src1) | IsQNaN(src2)

   if src1 >= src2 then do
      vresult.word[i] ← 0xFFFF_FFFF
      all_false       ← 0b0
   end
   else do
      vresult.word[i] ← 0x0000_0000
      all_true        ← 0b0
   end

   if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
   if vxvc_flag=1   then SetFX(FPSCR.VXVC)

   ex_flag ← ex_flag | (FPSCR.VE & vxsnan_flag)
                     | (FPSCR.VE & vxvc_flag)
end

if ex_flag=0 then VSR[32×TX+T] ← vresult

if Rc=1 then do
   if vex_flag=0 then
      CR.field[6] ← all_true || 0b0 || all_false ||
0b0
   else
      CR.field[6] ← 0bUUUU
end
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

For each integer value i from 0 to 3, do the following.

Let src1 be the single-precision floating-point operand in word element i of VSR[XA].

Let src2 be the single-precision floating-point operand in word element i of VSR[XB].

src1 is compared to src2.

The contents of word element i of VSR[XT] are set to all 1s if src1 is greater than or equal to src2, and is set to all 0s otherwise.

A NaN input causes the comparison to return false for that element.

Two zero inputs of same or different signs return true for that element.

Two infinity inputs of same signs return true for that element.

If Rc=1, CR Field 6 is set as follows.

- Bit 0 is set to indicate all vector elements compared true.
- Bit 1 is set to 0.
- Bit 2 is set to indicate all vector elements compared false.
- Bit 3 is set to 0.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT] and the contents of CR field 6 are undefined if Rc is equal to 1.

Special Registers Altered:

Register	*Field(s)*	
CR	CR6	(if Rc=1)
FPSCR	FX VXSNAN VXVC	

VSR Data Layout for xvcmpgesp[.]

src1	VSR[XA].word[0]	VSR[XA].word[1]	VSR[XA].word[2]	VSR[XA].word[3]
src2	VSR[XB].word[0]	VSR[XB].word[1]	VSR[XB].word[2]	VSR[XB].word[3]

tgt	VSR[XT].word[0]	VSR[XT].word[1]	VSR[XT].word[2]	VSR[XT].word[3]
	0	32	64	96 127

VSX Vector Compare Greater Than Double-Precision XX3-form

xvcmpgtdp	XT,XA,XB	(Rc=0)
xvcmpgtdp.	XT,XA,XB	(Rc=1)

60	T	A	B	Rc	107	AX	BX	TX
0	6	11	16	21	22	29	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

ex_flag   ← 0b0
all_false ← 0b1
all_true  ← 0b1

do i = 0 to 1
   reset_xflags()

   src1 ← bfp_CONVERT_FROM_BFP64(VSR[32×AX+A].dword[i])
   src2 ← bfp_CONVERT_FROM_BFP64(VSR[32×BX+B].dword[i])

   if src1.class.SNaN | src2.class.SNaN then do
      vxsnan_flag ← 0b1
      if FPSCR.VE=0 then vxvc_flag ← 0b1
   end
   else vxvc_flag ← IsQNaN(src1) | IsQNaN(src2)

   if src1 > src2 then do
      vresult.dword[i] ← 0xFFFF_FFFF_FFFF_FFFF
      all_false        ← 0b0
   end
   else do
      vresult.dword[i] ← 0x0000_0000_0000_0000
      all_true         ← 0b0
   end

   if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
   if vxvc_flag=1   then SetFX(FPSCR.VXVC)

   ex_flag ← ex_flag | (FPSCR.VE & vxsnan_flag)
                     | (FPSCR.VE & vxvc_flag)
end

if ex_flag=0 then VSR[32×TX+T] ← vresult

if Rc=1 then do
   if vex_flag=0 then
      CR.field[6] ← all_true || 0b0 || all_false ||
0b0
   else
      CR.field[6] ← 0bUUUU
end
```

Let `XT` be the value `32×TX + T`.
Let `XA` be the value `32×AX + A`.
Let `XB` be the value `32×BX + B`.

For each integer value `i` from 0 to 1, do the following.

Let `src1` be the double-precision floating-point operand in doubleword element `i` of `VSR[XA]`.

Let `src2` be the double-precision floating-point operand in doubleword element `i` of `VSR[XB]`.

`src1` is compared to `src2`.

The contents of doubleword element `i` of `VSR[XT]` are set to all 1s if `src1` is greater than `src2`, and is set to all 0s otherwise.

A NaN input causes the comparison to return false for that element.

Two zero inputs of same or different signs return false for that element.

If `Rc=1`, CR Field 6 is set as follows.

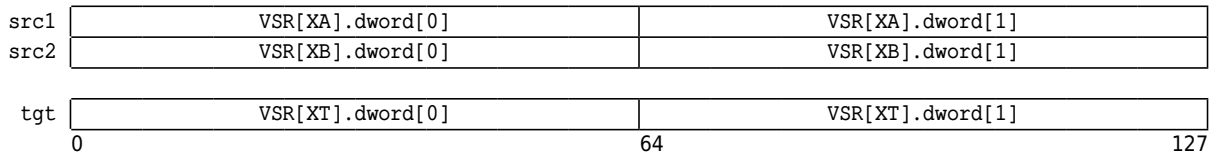
- Bit 0 is set to indicate all vector elements compared true.
- Bit 1 is set to 0.
- Bit 2 is set to indicate all vector elements compared false.
- Bit 3 is set to 0.

If a trap-enabled exception occurs in any element of the vector, no results are written to `VSR[XT]` and the contents of CR field 6 are undefined if `Rc` is equal to 1.

Special Registers Altered:

Register	*Field(s)*	
CR	CR6	(if Rc=1)
FPSCR	FX VXSNAN VXVC	

VSR Data Layout for xvcmpgtdp[.]



VSX Vector Compare Greater Than Single-Precision XX3-form

xvcmpgtsp	XT,XA,XB	(Rc=0)
xvcmpgtsp.	XT,XA,XB	(Rc=1)

60	T	A	B	Rc	75	AX	BX	TX
0	6	11	16	21	22	29	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

ex_flag   ← 0b0
all_false ← 0b1
all_true  ← 0b1

do i = 0 to 3
   reset_xflags()

   src1 ← bfp_CONVERT_FROM_BFP32(VSR[32×AX+A].word[i])
   src2 ← bfp_CONVERT_FROM_BFP32(VSR[32×BX+B].word[i])

   if IsSNaN(src1)=1 | IsSNaN(src2)=1 then do
      vxsnan_flag ← 0b1
      if FPSCR.VE=0 then vxvc_flag ← 0b1
   end
   else
      vxvc_flag ← src1.class.QNaN | src2.class.QNaN

   if src1 > src2 then do
      vresult.word[i] ← 0xFFFF_FFFF
      all_false       ← 0b0
   end
   else do
      vresult.word[i] ← 0x0000_0000
      all_true        ← 0b0
   end

   if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
   if vxvc_flag=1   then SetFX(FPSCR.VXVC)

   ex_flag ← ex_flag | (FPSCR.VE & vxsnan_flag)
                     | (FPSCR.VE & vxvc_flag)
end

if ex_flag=0 then VSR[32×TX+T] ← vresult

if Rc=1 then do
   if vex_flag=0 then
      CR.field[6] ← all_true || 0b0 || all_false ||
0b0
   else
      CR.field[6] ← 0bUUUU
end
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

For each integer value i from 0 to 3, do the following.

Let src1 be the single-precision floating-point operand in word element i of VSR[XA].

Let src2 be the single-precision floating-point operand in word element i of VSR[XB].

src1 is compared to src2.

The contents of word element i of VSR[XT] are set to all 1s if src1 is greater than src2, and is set to all 0s otherwise.

A NaN input causes the comparison to return false for that element.

Two zero inputs of same or different signs return false for that element.

If Rc=1, CR Field 6 is set as follows.

- Bit 0 is set to indicate all vector elements compared true.
- Bit 1 is set to 0.
- Bit 2 is set to indicate all vector elements compared false.
- Bit 3 is set to 0.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT] and the contents of CR field 6 are undefined if Rc is equal to 1.

Special Registers Altered:

Register	*Field(s)*	
CR	CR6	(if Rc=1)
FPSCR	FX VXSNAN VXVC	

VSR Data Layout for xvcmpgtsp[.]

src1	VSR[XA].word[0]	VSR[XA].word[1]	VSR[XA].word[2]	VSR[XA].word[3]
src2	VSR[XB].word[0]	VSR[XB].word[1]	VSR[XB].word[2]	VSR[XB].word[3]

tgt	VSR[XT].word[0]	VSR[XT].word[1]	VSR[XT].word[2]	VSR[XT].word[3]
	0	32	64	96 127

VSX Vector Maximum Double-Precision XX3-form

xvmaxdp XT,XA,XB

60	T	A	B	224	AX	BX	TX
0	6	11	16	21	29	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

ex_flag ← 0b0

do i = 0 to 1
   reset_xflags()

   src1 ← VSR[32×AX+A].dword[i]
   src2 ← VSR[32×BX+B].dword[i]
   vresult.dword[i] ← bfp64_MAXIMUM(src1,src2)

   if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)

   ex_flag ← ex_flag | (FPSCR.VE & vxsnan_flag)
end

if ex_flag=0 then VSR[32×TX+T] ← vresult
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

For each integer value i from 0 to 1, do the following.

Let src1 be the double-precision floating-point operand in doubleword element i of VSR[XA].

Let src2 be the double-precision floating-point operand in doubleword element i of VSR[XB].

If src1 is greater than src2, src1 is placed into doubleword element i of VSR[XT] in double-precision format. Otherwise, src2 is placed into doubleword element i of VSR[XT] in double-precision format.

The maximum of +0 and −0 is +0. The maximum of a QNaN and any value is that value. The maximum of any value and an SNaN when VE=0 is that SNaN converted to a QNaN.

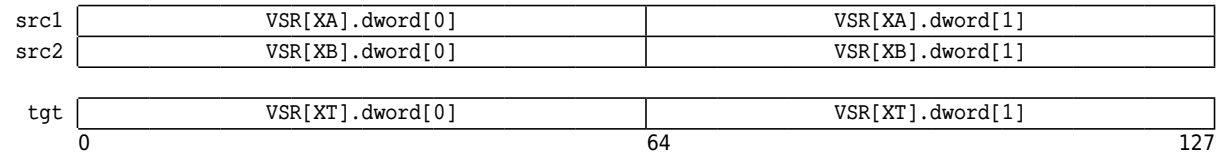
See Table 7.70.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered:

Register	*Field(s)*
FPSCR	FX VXSNAN

VSR Data Layout for xvmaxdp



src1 \ src2	−Infinity	−NZF	−Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
−Infinity	T(src1)	T(src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(src1)	T(Q(src2)) fx(VXSNAN)
−NZF	T(src1)	T(M(src1,src2))	T(src2)	T(src2)	T(src2)	T(src2)	T(src1)	T(Q(src2)) fx(VXSNAN)
−Zero	T(src1)	T(src1)	T(src1)	T(src2)	T(src2)	T(src2)	T(src1)	T(Q(src2)) fx(VXSNAN)
+Zero	T(src1)	T(src1)	T(src1)	T(src1)	T(src2)	T(src2)	T(src1)	T(Q(src2)) fx(VXSNAN)
+NZF	T(src1)	T(src1)	T(src1)	T(src1)	T(M(src1,src2))	T(src2)	T(src1)	T(Q(src2)) fx(VXSNAN)
+Infinity	T(src1)	T(src1)	T(src1)	T(src1)	T(src1)	T(src1)	T(src1)	T(Q(src2)) fx(VXSNAN)
QNaN	T(src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(src1)	T(src1) fx(VXSNAN)
SNaN	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)

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}).
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.
fx(x)	If x is equal to 0, FX is set to 1. x is set to 1.

Table 7.70: Actions for xvmaxdp

VSX Vector Maximum Single-Precision XX3-form

xvmaxsp XT,XA,XB

60	T	A	B	192	AX	BX	TX
0	6	11	16	21	29	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

ex_flag ← 0b0

do i = 0 to 3
   reset_xflags()

   src1 ← VSR[32×AX+A].word[i]
   src2 ← VSR[32×BX+B].word[i]
   vresult.word[i] ← bfp32_MAXIMUM(src1,src2)

   if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)

   ex_flag ← ex_flag | (FPSCR.VE & vxsnan_flag)
end

if ex_flag=0 then VSR[32×TX+T] ← vresult
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

For each integer value i from 0 to 3, do the following.

Let src1 be the single-precision floating-point operand in word element i of VSR[XA].

Let src2 be the single-precision floating-point operand in word element i of VSR[XB].

If src1 is greater than src2, src1 is placed into word element i of VSR[XT] in single-precision format. Otherwise, src2 is placed into word element i of VSR[XT] in single-precision format.

The maximum of +0 and −0 is +0. The maximum of a QNaN and any value is that value. The maximum of any value and an SNaN when VE=0 is that SNaN converted to a QNaN.

See Table 7.71.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered:

Register	*Field(s)*
FPSCR	FX VXSNAN

VSR Data Layout for xvmaxsp

src1	VSR[XA].word[0]	VSR[XA].word[1]	VSR[XA].word[2]	VSR[XA].word[3]
src2	VSR[XB].word[0]	VSR[XB].word[1]	VSR[XB].word[2]	VSR[XB].word[3]
tgt	VSR[XT].word[0]	VSR[XT].word[1]	VSR[XT].word[2]	VSR[XT].word[3]
	0	32	64	96 127

		src2							
		−Infinity	**−NZF**	**−Zero**	**+Zero**	**+NZF**	**+Infinity**	**QNaN**	**SNaN**
src1	**−Infinity**	T(src1)	T(src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(src1)	T(Q(src2)) fx(VXSNAN)
	−NZF	T(src1)	T(M(src1,src2))	T(src2)	T(src2)	T(src2)	T(src2)	T(src1)	T(Q(src2)) fx(VXSNAN)
	−Zero	T(src1)	T(src1)	T(src1)	T(src2)	T(src2)	T(src2)	T(src1)	T(Q(src2)) fx(VXSNAN)
	+Zero	T(src1)	T(src1)	T(src1)	T(src1)	T(src2)	T(src2)	T(src1)	T(Q(src2)) fx(VXSNAN)
	+NZF	T(src1)	T(src1)	T(src1)	T(src1)	T(M(src1,src2))	T(src2)	T(src1)	T(Q(src2)) fx(VXSNAN)
	+Infinity	T(src1)	T(src1)	T(src1)	T(src1)	T(src1)	T(src1)	T(src1)	T(Q(src2)) fx(VXSNAN)
	QNaN	T(src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(src1)	T(src1) fx(VXSNAN)
	SNaN	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)

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}`).
NZF	Nonzero finite number.
Q(x)	Return a QNaN with the payload of x.
M(x,y)	Return the greater of floating-point value x and floating-point value y.
T(x)	The value x is placed in word element `i` (`i`∈`{0,1,2,3}`) of `VSR[XT]` in single-precision format.
fx(x)	If FPSCR.x is equal to 0, FPSCR.FX is set to 1. FPSCR.x is set to 1.

Table 7.71: Actions for xvmaxsp

VSX Vector Minimum Double-Precision XX3-form

xvmindp XT,XA,XB

60	T	A	B	232	AX	BX	TX
0	6	11	16	21	29	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

ex_flag ← 0b0

do i = 0 to 1
   reset_xflags()

   src1 ← VSR[32×AX+A].dword[i]
   src2 ← VSR[32×BX+B].dword[i]
   vresult.dword[i] ← bfp64_MINIMUM(src1,src2)

   if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)

   ex_flag ← ex_flag | (FPSCR.VE & vxsnan_flag)
end

if ex_flag=0 then VSR[32×TX+T] ← vresult
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

For each integer value i from 0 to 1, do the following.

Let src1 be the double-precision floating-point operand in doubleword element i of VSR[XA].

Let src2 be the double-precision floating-point operand in doubleword element i of VSR[XB].

If src1 is less than src2, src1 is placed into doubleword element i of VSR[XT] in double-precision format. Otherwise, src2 is placed into doubleword element i of VSR[XT] in double-precision format.

The minimum of +0 and −0 is −0. The minimum of a QNaN and any value is that value. The minimum of any value and an SNaN when VE=0 is that SNaN converted to a QNaN.

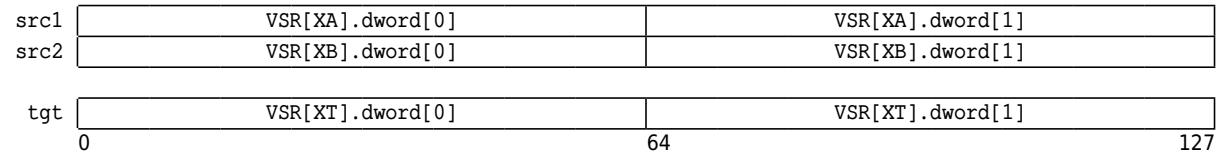
See Table 7.72.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered:

Register	*Field(s)*
FPSCR	FX VXSNAN

VSR Data Layout for xvmindp



		src2							
		−Infinity	**−NZF**	**−Zero**	**+Zero**	**+NZF**	**+Infinity**	**QNaN**	**SNaN**
src1	**−Infinity**	T(src1)	T(src1)	T(src1)	T(src1)	T(src1)	T(src1)	T(src1)	T(Q(src2)) fx(VXSNAN)
	−NZF	T(src2)	T(M(src1,src2))	T(src1)	T(src1)	T(src1)	T(src1)	T(src1)	T(Q(src2)) fx(VXSNAN)
	−Zero	T(src2)	T(src2)	T(src1)	T(src1)	T(src1)	T(src1)	T(src1)	T(Q(src2)) fx(VXSNAN)
	+Zero	T(src2)	T(src2)	T(src2)	T(src1)	T(src1)	T(src1)	T(src1)	T(Q(src2)) fx(VXSNAN)
	+NZF	T(src2)	T(src2)	T(src2)	T(src2)	T(M(src1,src2))	T(src1)	T(src1)	T(Q(src2)) fx(VXSNAN)
	+Infinity	T(src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(src1)	T(src1)	T(Q(src2)) fx(VXSNAN)
	QNaN	T(src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(src1)	T(src1) fx(VXSNAN)
	SNaN	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)

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.
fx(x)	If FPSCR.x is equal to 0, FPSCR.FX is set to 1. FPSCR.x is set to 1.

Table 7.72: Actions for xvmindp

VSX Vector Minimum Single-Precision XX3-form

xvminsp XT,XA,XB

60	T	A	B	200	AX	BX	TX
0	6	11	16	21	29	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

ex_flag ← 0b0

do i = 0 to 3
   reset_xflags()

   src1 ← VSR[32×AX+A].word[i]
   src2 ← VSR[32×BX+B].word[i]
   vresult.word[i] ← bfp32_MINIMUM(src1,src2)

   if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)

   ex_flag ← ex_flag | (FPSCR.VE & vxsnan_flag)
end

if ex_flag=0 then VSR[32×TX+T] ← vresult
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

For each integer value i from 0 to 3, do the following.

Let src1 be the single-precision floating-point operand in word element i of VSR[XA].

Let src2 be the single-precision floating-point operand in word element i of VSR[XB].

If src1 is less than src2, src1 is placed into word element i of VSR[XT] in single-precision format. Otherwise, src2 is placed into word element i of VSR[XT] in single-precision format.

The minimum of +0 and −0 is −0. The minimum of a QNaN and any value is that value. The minimum of any value and an SNaN when VE=0 is that SNaN converted to a QNaN.

See Table 7.73.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered:

Register	*Field(s)*
FPSCR	FX VXSNAN

VSR Data Layout for xvminsp

src1	VSR[XA].word[0]	VSR[XA].word[1]	VSR[XA].word[2]	VSR[XA].word[3]
src2	VSR[XB].word[0]	VSR[XB].word[1]	VSR[XB].word[2]	VSR[XB].word[3]
tgt	VSR[XT].word[0]	VSR[XT].word[1]	VSR[XT].word[2]	VSR[XT].word[3]
	0	32	64	96 127

		src2							
		−Infinity	**−NZF**	**−Zero**	**+Zero**	**+NZF**	**+Infinity**	**QNaN**	**SNaN**
src1	**−Infinity**	T(src1)	T(src1)	T(src1)	T(src1)	T(src1)	T(src1)	T(src1)	T(Q(src2)) fx(VXSNAN)
	−NZF	T(src2)	T(M(src1,src2))	T(src1)	T(src1)	T(src1)	T(src1)	T(src1)	T(Q(src2)) fx(VXSNAN)
	−Zero	T(src2)	T(src2)	T(src1)	T(src1)	T(src1)	T(src1)	T(src1)	T(Q(src2)) fx(VXSNAN)
	+Zero	T(src2)	T(src2)	T(src2)	T(src1)	T(src1)	T(src1)	T(src1)	T(Q(src2)) fx(VXSNAN)
	+NZF	T(src2)	T(src2)	T(src2)	T(src2)	T(M(src1,src2))	T(src1)	T(src1)	T(Q(src2)) fx(VXSNAN)
	+Infinity	T(src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(src1)	T(src1)	T(Q(src2)) fx(VXSNAN)
	QNaN	T(src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(src1)	T(src1) fx(VXSNAN)
	SNaN	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)

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}).
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`∈{0,1,2,3}) of `VSR[XT]` in single-precision format.
fx(x)	If FPSCR.x is equal to 0, FPSCR.FX is set to 1. FPSCR.x is set to 1.
VXSNAN	Floating-point Invalid Operation Exception (SNaN). If FPSCR.VE=1, update of `VSR[XT]` is suppressed.

Table 7.73: Actions for xvminsp

7.6.2.5 VSX Binary Floating-Point Round to Shorter Precision Instructions

VSX Scalar Round Quad-Precision to Double-Extended-Precision Z23-form

xsrqpxp R,VRT,VRB,RMC

63	VRT	///	R	VRB	RMC	37	/
0	6	11	15	16	21	23	31

```
if MSR.VSX=0 then VSX_Unavailable()

reset_xflags()

if R=0 then do
   if RMC=0b00 then           // Round to Nearest Away
      rmode ← 0b100
   if RMC=0b11 then do
      if FPSCR.RN=0b00 then   // Round to Nearest Even
         rmode ← 0b000
      if FPSCR.RN=0b01 then   // Round towards Zero
         rmode ← 0b001
      if FPSCR.RN=0b10 then   // Round towards +Infinity
         rmode ← 0b010
      if FPSCR.RN=0b11 then   // Round towards -Infinity
         rmode ← 0b011
   end
end
else do // R=1
   if RMC=0b00 then           // Round to Nearest Even
      rmode ← 0b000
   if RMC=0b01 then           // Round towards Zero
      rmode ← 0b001
   if RMC=0b10 then           // Round towards +Infinity
      rmode ← 0b010
   if RMC=0b11 then           // Round towards -Infinity
      rmode ← 0b011
end

src    ← bfp_CONVERT_FROM_BFP128(VSR[VRB+32])
rnd    ← bfp_ROUND_TO_BFP80(rmode,src)
result ← bfp128_CONVERT_FROM_BFP(rnd)

if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
if ox_flag=1 then SetFX(FPSCR.OX)
if ux_flag=1 then SetFX(FPSCR.UX)
if xx_flag=1 then SetFX(FPSCR.XX)

vex_flag ← FPSCR.VE & vxsnan_flag

if vex_flag=0 then do
   VSR[VRT+32] ← result
   FPSCR.FPRF  ← fprf_CLASS_BFP128(result)
end
FPSCR.FR ← (vxsnan_flag=0) & inc_flag
FPSCR.FI ← (vxsnan_flag=0) & xx_flag
```

Let R and RMC specify the rounding mode as follows.

R	RMC	FPSCR.RN	Rounding Mode
0	00	–	Round to Nearest Away
0	01	–	reserved
0	10	–	reserved
0	11	00	Round to Nearest Even
0	11	01	Round to Zero
0	11	10	Round to +Infinity
0	11	11	Round to -Infinity
1	00	–	Round to Nearest Even
1	01	–	Round to Zero
1	10	–	Round to +Infinity
1	11	–	Round to -Infinity

Let src be the floating-point value in VSR[VRB+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 7.9, “Scalar Floating-Point Intermediate Result Handling,” on page 617.

The result is placed into VSR[VRT+32] in quad-precision format.

FPRF is set to the class and sign of the result. FR is set to indicate if the rounded result was incremented. FI is set to indicate the result is inexact.

If a trap-disabled Invalid Operation exception occurs, FPRF is set to an undefined value, and FR and FI are set to 0.

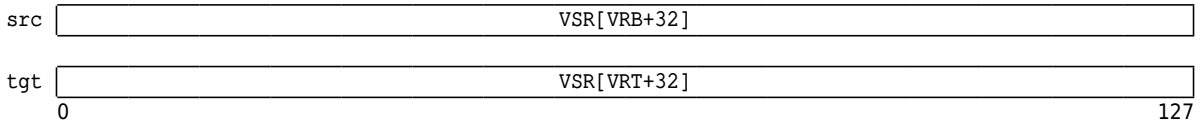
If a trap-enabled Invalid Operation exception occurs, VSR[VRT+32] and FPRF are not modified, and FR and FI are set to 0.

See Table 7.10, “VSX Scalar Floating-Point Final Result,” on page 618.

Special Registers Altered:

Register	*Field(s)*
FPSCR	FPRF FR FI FX VXSNAN OX UX XX

VSR Data Layout for xsrqpxp



VSX Scalar Round to Single-Precision XX2-form

xsrsp XT,XB

60	T	///	B	281	BX	TX
0	6	11	16	21	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

reset_xflags()

src      ← bfp_CONVERT_FROM_BFP64(VSR[32×BX+B].dword[0])
rnd      ← bfp_ROUND_TO_BFP32(FPSCR.RN,src)
result32 ← bfp32_CONVERT_FROM_BFP(rnd)
result64 ← bfp64_CONVERT_FROM_BFP(rnd)

if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
if ox_flag=1 then SetFX(FPSCR.OX)
if ux_flag=1 then SetFX(FPSCR.UX)
if xx_flag=1 then SetFX(FPSCR.XX)

vex_flag ← FPSCR.VE & vxsnan_flag

if vex_flag=0 then do
   VSR[32×TX+T].dword[0] ← result64
   VSR[32×TX+T].dword[1] ← 0x0000_0000_0000_0000
   FPSCR.FPRF ← fprf_CLASS_BFP32(result32)
   FPSCR.FR   ← inc_flag
   FPSCR.FI   ← xx_flag
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].

src is rounded to single-precision using the rounding mode specified by RN.

See Table 7.9, “Scalar Floating-Point Intermediate Result Handling,” on page 617.

The result is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are set to 0.

FPRF is set to the class and sign of the result as represented in single-precision format.

If a trap-enabled invalid operation exception occurs, VSR[XT] and FPRF are not modified.

Special Registers Altered:

Register	*Field(s)*
FPSCR	FPRF FR FI FX OX UX XX VXSNAN

Programming Note

Previous versions of the architecture allowed the contents of doubleword 1 of the result register to be undefined. However, all processors that support this instruction write 0s into doubleword 1 of the result register, as is required by this version of the architecture.

VSR Data Layout for xsrsp

7.6.2.6 VSX Binary Floating-Point Convert to Shorter Precision Instructions

VSX Scalar Convert with round Double-Precision to Half-Precision format XX2-form

xscvdphp XT,XB

60	T	17	B	347	BX	TX
0	6	11	16	21	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

reset_flags()

src    ← bfp_CONVERT_FROM_BFP64(VSR[BX×32+B].dword[0])
rnd    ← bfp_ROUND_TO_BFP16(FPSCR.RN,src)
result ← bfp16_CONVERT_FROM_BFP(rnd)

if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
if ox_flag=1 then SetFX(FPSCR.OX)
if ux_flag=1 then SetFX(FPSCR.UX)
if xx_flag=1 then SetFX(FPSCR.XX)

vex_flag ← FPSCR.VE & vxsnan_flag

if vex_flag=0 then do
   VSR[TX×32+T].hword[0:2] ← 0x0000_0000_0000
   VSR[TX×32+T].hword[3]   ← result
   VSR[TX×32+T].dword[1]   ← 0x0000_0000_0000_0000
   FPSCR.FPRF ← fprf_CLASS_BFP16(result)
end
FPSCR.FR ← (vex_flag=0) & inc_flag
FPSCR.FI ← (vex_flag=0) & xx_flag
```

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 an SNaN, the result is the half-precision representation of that SNaN converted to a QNaN.

Otherwise, if src is a QNaN, the result is the half-precision representation of that QNaN.

Otherwise, if src is an Infinity, the result is the half-precision representation of Infinity with the same sign as src.

Otherwise, if src is a Zero, the result is the half-precision representation of Zero with the same sign as src.

Otherwise, the result is the half-precision representation of src rounded to half-precision using the rounding mode specified by RN.

The result is zero-extended and placed into doubleword element 0 of VSR[XT].

The contents of doubleword element 1 of VSR[XT] are set to 0.

FPRF is set to the class and sign of the result as represented in half-precision. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.

If a trap-enabled invalid operation exception occurs, VSR[XT] and FPRF are not modified, and FR and FI are set to 0.

Special Registers Altered:

Register	*Field(s)*
FPSCR	FPRF FR FI FX VXSNAN OX UX XX

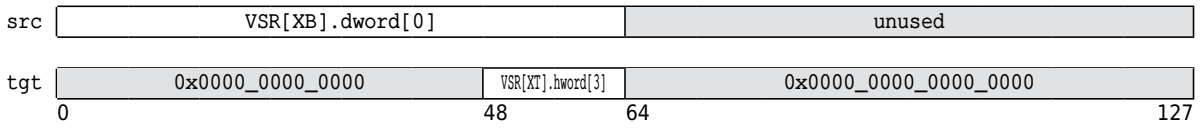
Programming Note

This instruction can be used to operate on a single-precision source operand.

Programming Note

Previous versions of the architecture allowed the contents of doubleword 1 of the result register to be undefined. However, all processors that support this instruction write 0s into doubleword 1 of the result register, as is required by this version of the architecture.

VSR Data Layout for xscvdphp



VSX Scalar Convert with round Double-Precision to Single-Precision format XX2-form

xscvdpsp XT,XB

60	T	///	B	265	BX	TX
0	6	11	16	21	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

reset_xflags()

src    ← bfp_CONVERT_FROM_BFP64(VSR[32×BX+B].dword[0])
rnd    ← bfp_ROUND_TO_BFP32(FPSCR.RN,src)
result ← bfp32_CONVERT_FROM_BFP(rnd)

if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
if xx_flag=1 then SetFX(FPSCR.XX)
if ox_flag=1 then SetFX(FPSCR.OX)
if ux_flag=1 then SetFX(FPSCR.UX)

vex_flag ← FPSCR.VE & vxsnan_flag

if vex_flag=0 then do
   VSR[32×TX+T].word[0] ← result
   VSR[32×TX+T].word[1] ← result
   VSR[32×TX+T].word[2] ← 0x0000_0000
   VSR[32×TX+T].word[3] ← 0x0000_0000
   FPSCR.FPRF ← fprf_CLASS_BFP32(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 7.9, “Scalar Floating-Point Intermediate Result Handling,” on page 617.

The result is placed into word elements 0 and 1 of VSR[XT] in single-precision format.

The contents of word elements 2 and 3 of VSR[XT] are set to 0.

FPRF is set to the class and sign of the result. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.

If a trap-enabled invalid operation exception occurs, VSR[XT] and FPRF are not modified, and FR and FI are set to 0.

See Table 7.10, “VSX Scalar Floating-Point Final Result,” on page 618.

Special Registers Altered:

Register	*Field(s)*
FPSCR	FPRF FR FI FX OX UX XX VXSNAN

Programming Note

This instruction can be used to operate on a single-precision source operand.

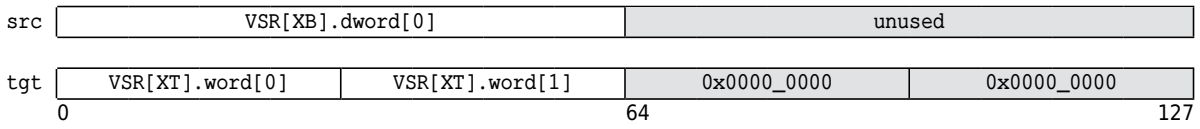
Programming Note

Previous versions of the architecture allowed the contents of words 1, 2, and 3 of the result register to be undefined, however, all processors that support this instruction write the result into both words 0 and 1 of the result register, as is required by this version of the architecture.

Programming Note

Previous versions of the architecture allowed the contents of doubleword 1 of the result register to be undefined. However, all processors that support this instruction write 0s into doubleword 1 of the result register, as is required by this version of the architecture.

VSR Data Layout for xscvdpsp



VSX Scalar Convert Scalar Single-Precision to Vector Single-Precision format Non-signalling XX2-form

xscvdpspn XT,XB

60	T	///	B	267	BX	TX
0	6	11	16	21	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

reset_xflags()

src    ← bfp_CONVERT_FROM_BFP64(VSR[32×BX+B].dword[0])
result ← bfp32_CONVERT_FROM_BFP(src)

VSR[32×TX+T].word[0] ← result
VSR[32×TX+T].word[1] ← result
VSR[32×TX+T].word[2] ← 0x0000_0000
VSR[32×TX+T].word[3] ← 0x0000_0000
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

Let src be the single-precision floating-point value in doubleword element 0 of VSR[XB] represented in double-precision format.

src is placed into word elements 0 and 1 of VSR[XT] in single-precision format.

The contents of word elements 2 and 3 of VSR[XT] are set to 0.

Special Registers Altered:

None

Programming Note

If x is not representable in single-precision, some exponent and/or significand bits will be discarded, likely producing undesirable results. The low-order 29 bits of the significand of x are discarded, more if the unbiased exponent of x is less than -126 (i.e., denormal). Finite values of x having an unbiased exponent less than -150 will return a result of Zero. Finite values of x having an unbiased exponent greater than +127 will result in discarding significant bits of the exponent. SNaN inputs having no significant bits in the upper 23 bits of the signifcand will return Infinity as the result. No status is set for any of these cases.

Programming Note

xscvdpsp should be used to convert a scalar double-precision value to vector single-precision format.

xscvdpspn should be used to convert a scalar single-precision value to vector single-precision format.

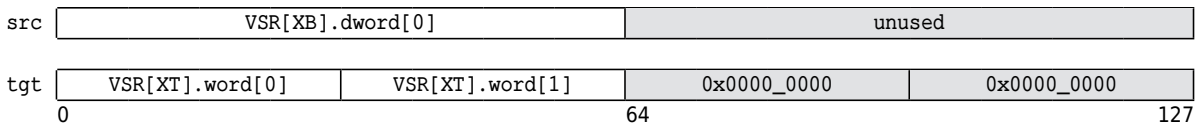
Programming Note

See the Programming Note for the ***xscvdpsp*** instruction.

Programming Note

Previous versions of the architecture allowed the contents of doubleword 1 of the result register to be undefined. However, all processors that support this instruction write 0s into doubleword 1 of the result register, as is required by this version of the architecture.

VSR Data Layout for xscvdpspn



VSX Scalar Convert with round Quad-Precision to Double-Precision format [using round to Odd] X-form

xscvqpdp	VRT,VRB	(RO=0)
xscvqpdpo	VRT,VRB	(RO=1)

63	VRT	20	VRB	836	RO
0	6	11	16	21	31

```
if MSR.VSX=0 then VSX_Unavailable()

reset_xflags()

src    ← bfp_CONVERT_FROM_BFP128(VSR[VRB+32])
rnd    ← bfp_ROUND_TO_BFP64(RO,FPSCR.RN,src)
result ← bfp64_CONVERT_FROM_BFP(rnd)

if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
if ox_flag=1 then SetFX(FPSCR.OX)
if ux_flag=1 then SetFX(FPSCR.UX)
if xx_flag=1 then SetFX(FPSCR.XX)

vex_flag ← FPSCR.VE & vxsnan_flag

if vex_flag=0 then do
   VSR[VRT+32].dword[0] ← result
   VSR[VRT+32].dword[1] ← 0x0000_0000_0000_0000
   FPSCR.FPRF ← fprf_CLASS_BFP64(result)
end
FPSCR.FR ← (vxsnan_flag=0) & inc_flag
FPSCR.FI ← (vxsnan_flag=0) & xx_flag
```

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 7.9, “Scalar Floating-Point Intermediate Result Handling,” on page 617.

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.

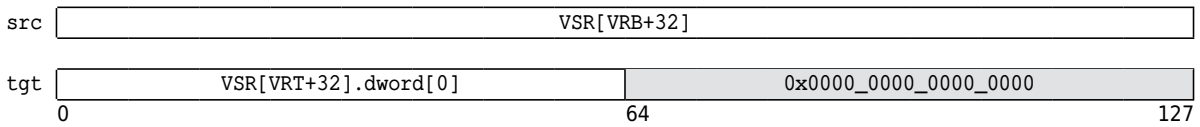
If a trap-enabled Invalid Operation exception occurs, VSR[VRT+32] and FPRF are not modified, and FR and FI are set to 0.

See Table 7.10, “VSX Scalar Floating-Point Final Result,” on page 618.

Special Registers Altered:

Register	*Field(s)*
FPSCR	FPRF FR FI FX VXSNAN OX UX XX

VSR Data Layout for xscvqpdp[o]



VSX Vector Convert with round Single-Precision to bfloat16 format XX2-form

xvcvspbf16 XT,XB

60	T	17	B	475	BX	TX
0	6	11	16	21	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

do i = 0 to 3
   reset_flags()

   src ← bfp_CONVERT_FROM_BFP32(VSR[32×BX+B].word[i])
   rnd ← bfp_ROUND_TO_BFLOAT16(FPSCR.RN,src)
   result.word[i].hword[0] ← 0x0000
   result.word[i].hword[1] ←
      bfloat16_CONVERT_FROM_BFP(rnd)

   if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
   if ox_flag=1 then SetFX(FPSCR.OX)
   if ux_flag=1 then SetFX(FPSCR.UX)
   if xx_flag=1 then SetFX(FPSCR.XX)

   ex_flag ← ex_flag | (FPSCR.VE & vxsnan_flag)
                     | (FPSCR.OE & ox_flag)
                     | (FPSCR.UE & ux_flag)
                     | (FPSCR.XE & xx_flag)
end

if ex_flag=0 then VSR[32×TX+T] ← result
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

For each integer value i from 0 to 3, do the following.

Let src be the single-precision floating-point value in word element i of VSR[XB].

If src is an SNaN, let result be the bfloat16 representation of that SNaN converted to a QNaN.

Otherwise, if src is a QNaN, let result be the bfloat16 representation of that QNaN.

Otherwise, if src is an Infinity, let result be the bfloat16 representation of Infinity with the same sign as src.

Otherwise, if src is a Zero, let result be the bfloat16 representation of Zero with the same sign as src.

Otherwise, let result be the bfloat16 representation of src rounded to bfloat16 precision using the rounding mode specified in RN.

result is placed into rightmost halfword of word element i of VSR[XT].

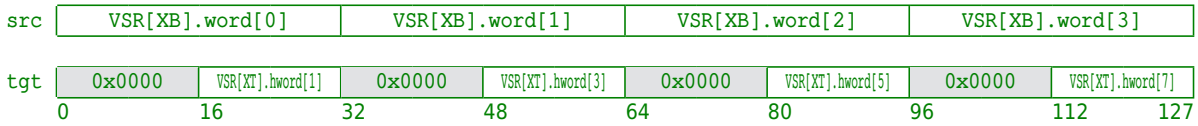
The leftmost halfword of word element i of VSR[XT] is set to 0x0000.

If a trap-enabled exception occurs, VSR[XT] is not modified.

Special Registers Altered:

Register	*Field(s)*
FPSCR	FX VXSNAN OX UX XX

Register Operand Data Layout for xvcvspbf16



VSX Vector Convert with round Double-Precision to Single-Precision format XX2-form

xvcvdpsp XT,XB

60	T	///	B	393	BX	TX
0	6	11	16	21	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

ex_flag ← 0b0

do i = 0 to 1
   reset_xflags()

   src ← bfp_CONVERT_FROM_BFP64(VSR[32×BX+B].dword[i])
   rnd ← bfp_ROUND_TO_BFP32(FPSCR.RN,src)

   vresult.dword[i].word[0] ←
      bfp32_CONVERT_FROM_BFP(rnd)
   vresult.dword[i].word[1] ←
      bfp32_CONVERT_FROM_BFP(rnd)

   if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
   if ox_flag=1 then SetFX(FPSCR.OX)
   if ux_flag=1 then SetFX(FPSCR.UX)
   if xx_flag=1 then SetFX(FPSCR.XX)

   ex_flag ← ex_flag | (FPSCR.VE & vxsnan_flag)
                     | (FPSCR.OE & ox_flag)
                     | (FPSCR.UE & ux_flag)
                     | (FPSCR.XE & xx_flag)
end

if ex_flag=0 then VSR[32×TX+T] ← result
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

For each integer value i from 0 to 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. If src is a Signalling NaN, the result is the corresponding Quiet NaN, and VXSNAN is set to 1.

The result is placed into bits 0:31 and bits 32:63 of doubleword element i of VSR[XT] in single-precision format.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered:

Register	*Field(s)*
FPSCR	FX OX UX XX VXSNAN

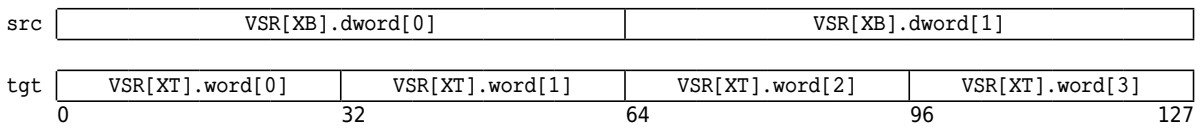
Programming Note

Previous versions of the architecture allowed the contents of bits 32:63 of each doubleword in the result register to be undefined, however, all processors that support this instruction write the result into bits 32:63 of each doubleword in the result register as well as into bits 0:31, as is required by this version of the architecture.

Programming Note

Previous versions of the architecture allowed the contents of doubleword 1 of the result register to be undefined. However, all processors that support this instruction write 0s into doubleword 1 of the result register, as is required by this version of the architecture.

VSR Data Layout for xvcvdpsp



VSX Vector Convert with round Single-Precision to Half-Precision format XX2-form

xvcvsphp XT,XB

60	T	25	B	475	BX	TX
0	6	11	16	21	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

reset_flags()

do i = 0 to 3
   src ← bfp_CONVERT_FROM_BFP32(VSR[BX×32+B].word[i])
   rnd ← bfp_ROUND_TO_BFP16(FPSCR.RN,src)

   vresult.word[i].hword[0] ← 0x0000
   vresult.word[i].hword[1] ←
      bfp16_CONVERT_FROM_BFP(rnd)

   if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
   if ox_flag=1 then SetFX(FPSCR.OX)
   if ux_flag=1 then SetFX(FPSCR.UX)
   if xx_flag=1 then SetFX(FPSCR.XX)

   ex_flag ← ex_flag | (FPSCR.VE & vxsnan_flag)
                     | (FPSCR.OE & ox_flag)
                     | (FPSCR.UE & ux_flag)
                     | (FPSCR.XE & xx_flag)
end

if ex_flag=0 then VSR[32×TX+T] ← vresult
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

For each integer value i from 0 to 3, do the following.

Let src be the single-precision floating-point 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:

Register	*Field(s)*
FPSCR	FX VXSNAN OX UX XX

VSR Data Layout for xvcvsphp

7.6.2.7 VSX Binary Floating-Point Convert to Longer Precision Instructions

VSX Scalar Convert Double-Precision to Quad-Precision format X-form

xscvdpqp VRT,VRB

63	VRT	22	VRB	836	/
0	6	11	16	21	31

```
if MSR.VSX=0 then VSX_Unavailable()

src ← bfp_CONVERT_FROM_BFP64(VSR[VRB+32].dword[0])

if src.class.SNaN then
   result ← bfp128_CONVERT_FROM_BFP(bfp_QUIET(src))
else
   result ← bfp128_CONVERT_FROM_BFP(src)

vxsnan_flag ← src.class.SNaN
if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
vex_flag ← FPSCR.VE & vxsnan_flag

if vex_flag=0 then do
   VSR[VRT+32] ← result
   FPSCR.FPRF  ← fprf_CLASS_BFP128(result)
end
FPSCR.FR ← 0
FPSCR.FI ← 0
```

Let src be the floating-point value in doubleword element 0 of VSR[VRB+32] represented in double-precision format.

If src is a Signalling NaN, an Invalid Operation exception occurs, VXSNAN is set to 1, and the Quiet NaN corresponding to src is placed into VSR[VRT+32] in quad-precision format.

Otherwise, src is placed into VSR[VRT+32] in quad-precision format.

FPRF is set to the class and sign of the result.

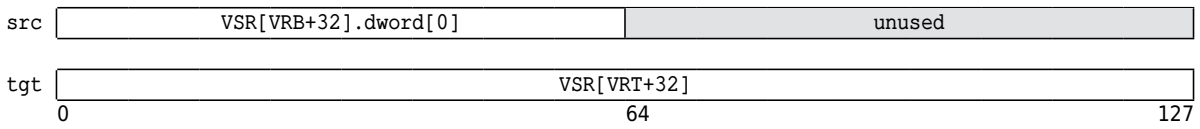
FR is set to 0. FI is set to 0.

If a trap-enabled Invalid Operation exception occurs, VSR[XT] and FPRF are not modified.

Special Registers Altered:

Register	*Field(s)*
FPSCR	FPRF FX VXSNAN
FPSCR	FR (set to 0)
FPSCR	FI (set to 0)

VSR Data Layout for xscvdpqp



VSX Scalar Convert Half-Precision to Double-Precision format XX2-form

xscvhpdp XT,XB

60	T	16	B	347	BX	TX
0	6	11	16	21	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

reset_flags()

src ← bfp_CONVERT_FROM_BFP16(VSR[BX×32+B].hword[3])

if src.class.SNaN=1 then
   result ← bfp64_CONVERT_FROM_BFP(bfp_QUIET(src))
else
   result ← bfp64_CONVERT_FROM_BFP(src)

vxsnan_flag ← src.class.SNaN
if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
vex_flag ← FPSCR.VE & vxsnan_flag

if vex_flag=0 then do
   VSR[TX×32+T].dword[0] ← result
   VSR[TX×32+T].dword[1] ← 0x0000_0000_0000_0000
   FPSCR.FPRF ← fprf_CLASS_BFP64(result)
end
FPSCR.FR ← 0
FPSCR.FI ← 0
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

Let src be the half-precision floating-point value in the rightmost halfword of doubleword element 0 of VSR[XB].

If src is an SNaN, the result is the double-precision representation of that SNaN converted to a QNaN.

Otherwise, if src is a QNaN, the result is the double-precision representation of that QNaN.

Otherwise, if src is an Infinity, the result is the double-precision representation of Infinity with the same sign as src.

Otherwise, if src is a Zero, the result is the double-precision representation of Zero with the same sign as src.

Otherwise, if src is a denormal value, the result is the normalized double-precision representation of src.

Otherwise, the result is the double-precision representation of src.

The result is placed into doubleword element 0 of VSR[XT].

The contents of doubleword element 1 of VSR[XT] are set to 0.

FPRF is set to the class and sign of the result as represented in double-precision format.

If a trap-enabled invalid operation exception occurs, VSR[XT] and FPRF are not modified.

FR is set to 0. FI is set to 0.

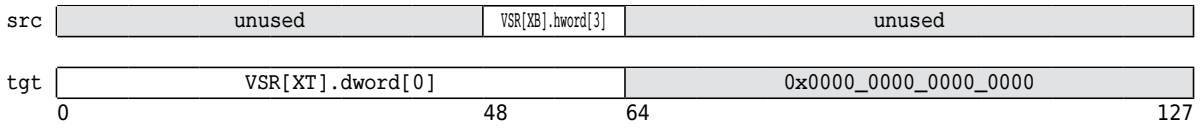
Special Registers Altered:

Register	*Field(s)*
FPSCR	FPRF FX VXSNAN
FPSCR	FR (set to 0)
FPSCR	FI (set to 0)

Programming Note

Previous versions of the architecture allowed the contents of doubleword 1 of the result register to be undefined. However, all processors that support this instruction write 0s into doubleword 1 of the result register, as is required by this version of the architecture.

VSR Data Layout for xscvhpdp



VSX Scalar Convert Single-Precision to Double-Precision format XX2-form

xscvspdp XT,XB

60	T	///	B	329	BX	TX
0	6	11	16	21	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

reset_xflags()

src ← bfp_CONVERT_FROM_BFP32(VSR[32×BX+B].word[0])
vxsnan_flag ← src.class.SNaN
result ← bfp64_CONVERT_FROM_BFP(src)
if vxsnan_flag=1 then do
   SetFX(FPSCR.VXSNAN)
   result ← bfp_QUIET(result)
end
vex_flag ← FPSCR.VE & vxsnan_flag
FPSCR.FR ← 0b0
FPSCR.FI ← 0b0
if vex_flag=0 then do
   VSR[32×TX+T].dword[0] ← result
   VSR[32×TX+T].dword[1] ← 0x0000_0000_0000_0000
   FPSCR.FPRF ← fprf_CLASS_BFP64(result)
end
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

Let src be the single-precision floating-point value in word element 0 of VSR[XB].

If src is a SNaN, the result is src, converted to a QNaN (i.e., bit 9 of src set to 1). VXSNAN is set to 1.

Otherwise, the result is src.

The result is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are set to 0.

FPRF is set to the class and sign of the result. FR is set to 0. FI is set to 0.

If a trap-enabled invalid operation exception occurs, VSR[XT] is not modified, FPRF is not modified, FR is set to 0, and FI is set to 0.

Special Registers Altered:

Register	*Field(s)*
FPSCR	FPRF FX VXSNAN
FPSCR	FR (set to 0)
FPSCR	FI (set to 0)

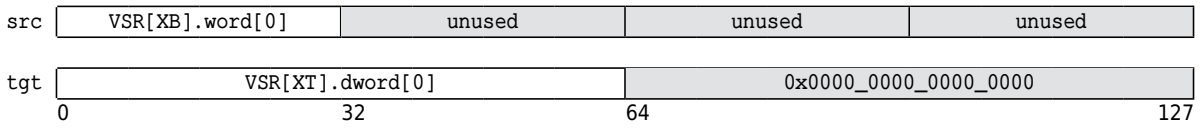
Programming Note

xscvspdp can be used to convert a single-precision value in single-precision format to double-precision format for use by Floating-Point scalar single-precision operations.

Programming Note

Previous versions of the architecture allowed the contents of doubleword 1 of the result register to be undefined. However, all processors that support this instruction write 0s into doubleword 1 of the result register, as is required by this version of the architecture.

VSR Data Layout for xscvspdp



VSX Scalar Convert Single-Precision to Double-Precision format Non-signalling XX2-form

xscvspdpn XT,XB

60	T	///	B	331	BX	TX
0	6	11	16	21	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

reset_xflags()

src ← bfp_CONVERT_FROM_BFP32(VSR[32×BX+B].word[0])
result ← bfp64_CONVERT_FROM_BFP(src)
VSR[32×TX+T].dword[0] ← result
VSR[32×TX+T].dword[1] ← 0x0000_0000_0000_0000
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

Let src be the single-precision floating-point value in word element 0 of VSR[XB].

src is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are set to 0.

Special Registers Altered:

None

Programming Note

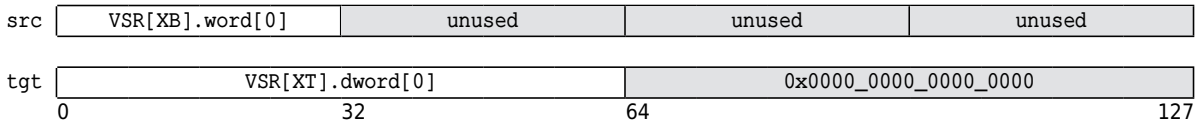
xscvspdp should be used to convert a vector single-precision floating-point value to scalar double-precision format.

xscvspdpn should be used to convert a vector single-precision floating-point value to scalar single-precision format.

Programming Note

Previous versions of the architecture allowed the contents of doubleword 1 of the result register to be undefined. However, all processors that support this instruction write 0s into doubleword 1 of the result register, as is required by this version of the architecture.

VSR Data Layout for xscvspdpn



VSX Vector Convert Half-Precision to Single-Precision format XX2-form

xvcvhpsp XT,XB

60	T	24	B	475	BX	TX
0	6	11	16	21	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

reset_flags()

do i = 0 to 3
   src ← bfp_CONVERT_FROM_BFP16(
               VSR[BX×32+B].word[i].hword[1])

   if src.class.SNaN=1 then
      vresult.word[i] ←
         bfp32_CONVERT_FROM_BFP(bfp_QUIET(src))
   else
      vresult.word[i] ← bfp32_CONVERT_FROM_BFP(src)

   vxsnan_flag ← src.class.SNaN
   if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
   ex_flag ← ex_flag | (FPSCR.VE & vxsnan_flag)
end

if ex_flag=0 then VSR[32×TX+T] ← vresult
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

For each integer value i from 0 to 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.

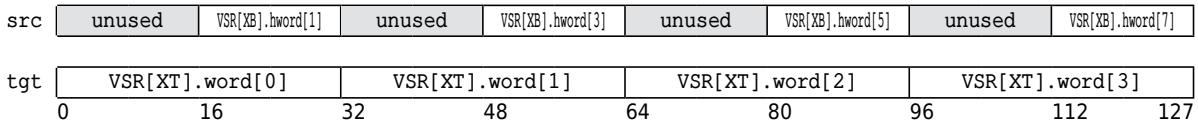
The result is placed into word element i of VSR[XT].

If a trap-enabled exception occurs, VSR[XT] is not modified.

Special Registers Altered:

Register	*Field(s)*
FPSCR	FX VXSNAN

VSR Data Layout for xvcvhpsp



VSX Vector Convert bfloat16 to Single-Precision format Non-signaling XX2-form

xvcvbf16spn XT,XB

60	T	16	B	475	BX	TX
0	6	11	16	21	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

reset_flags()

do i = 0 to 3
   VSR[32×TX+T].word[i].hword[0] ←
      VSR[32×BX+B].word[i].hword[1]
   VSR[32×TX+T].word[i].hword[1] ← 0x0000
end
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

For each integer value i from 0 to 3, do the following.

The contents of the rightmost halfword of word element i of VSR[XB] are placed into the leftmost halfword of word element i of VSR[XT].

The contents of the rightmost halfword of word element i of VSR[XT] are set to 0.

Special Registers Altered:

None

VSX Vector Convert Single-Precision to Double-Precision format XX2-form

xvcvspdp XT,XB

60	T	///	B	457	BX	TX
0	6	11	16	21	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

ex_flag ← 0b0
do i = 0 to 1
   reset_xflags()

   src ← bfp_CONVERT_FROM_BFP32(
               VSR[BX×32+B].dword[i].word[0])

   vxsnan_flag ← src.class.SNaN
   if src.class.SNaN=1 then src ← bfp_QUIET(src)
   vresult.dword[i] ← bfp64_CONVERT_FROM_BFP(src)

   if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
   ex_flag ← ex_flag | (FPSCR.VE & vxsnan_flag)
end

if ex_flag=0 then VSR[32×TX+T] ← vresult
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

For each integer value i from 0 to 1, do the following.

Let src be the single-precision floating-point operand in bits 0:31 of doubleword element i of VSR[XB].

src is placed into doubleword element i of VSR[XT] in double-precision format. If src is a Signalling NaN, the value placed into VSR[XT] is the corresponding Quiet NaN in double-precision format, and VXSNAN is set to 1.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered:

Register	*Field(s)*
FPSCR	FX VXSNAN

Register Operand Data Layout for xvcvbf16spn

src	unused	VSR[XB].hword[1]	unused	VSR[XB].hword[3]	unused	VSR[XB].hword[5]	unused	VSR[XB].hword[7]
tgt	VSR[XT].word[0]		VSR[XT].word[1]		VSR[XT].word[2]		VSR[XT].word[3]	

0 16 32 48 64 80 96 112 127

VSR Data Layout for xvcvspdp

src	VSR[XB].word[0]	unused	VSR[XB].word[2]	unused
tgt	VSR[XT].dword[0]		VSR[XT].dword[1]	

0 32 64 96 127

7.6.2.8 VSX Binary Floating-Point Round to Integral Instructions

7.6.2.8.1 VSX Scalar BFP Round to Integral Instructions

VSX Scalar Round to Double-Precision Integer using round to Nearest Away XX2-form

xsrdpi XT,XB

60	T	///	B	73	BX	TX
0	6	11	16	21	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

reset_xflags()

src    ← bfp_CONVERT_FROM_BFP64(VSR[VRB+32].dword[0])
rnd    ← bfp_ROUND_TO_INTEGER(0b100, src)
result ← bfp64_CONVERT_FROM_BFP(rnd)

if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)

vex_flag ← FPSCR.VE & vxsnan_flag

if vex_flag=0 then do
   VSR[32×TX+T].dword[0] ← result
   VSR[32×TX+T].dword[1] ← 0x0000_0000_0000_0000
   FPSCR.FPRF ← fprf_CLASS_BFP64(result)
end
FPSCR.FR ← 0b0
FPSCR.FI ← 0b0
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

Let src be the double-precision floating-point value in doubleword element 0 of VSR[XB].

src is rounded to an integer using the rounding mode Round to Nearest Away. If src is a Signalling NaN, the result is the corresponding Quiet NaN, and VXSNAN is set to 1.

The result is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are set to 0.

FPRF is set to the class and sign of the result. FR is set to 0. FI is set to 0.

If a trap-enabled invalid operation exception occurs, VSR[XT] and FPRF are not modified, and FR and FI are set to 0.

Special Registers Altered:

Register	*Field(s)*
FPSCR	FPRF FX VXSNAN
FPSCR	FR (set to 0)
FPSCR	FI (set to 0)

Programming Note

This instruction can be used to operate on a single-precision source operand.

Programming Note

Previous versions of the architecture allowed the contents of doubleword 1 of the result register to be undefined. However, all processors that support this instruction write 0s into doubleword 1 of the result register, as is required by this version of the architecture.

VSR Data Layout for xsrdpi

VSX Scalar Round to Double-Precision Integer exact using Current rounding mode XX2-form

xsrdpic XT,XB

60	T	///	B	107	BX	TX
0	6	11	16	21	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

reset_xflags()

src ← bfp_CONVERT_FROM_BFP64(VSR[VRB+32].dword[0])

if FPSCR.RN=0b00 then
   rnd ← bfp_ROUND_TO_INTEGER(0b000, src)
if FPSCR.RN=0b01 then
   rnd ← bfp_ROUND_TO_INTEGER(0b001, src)
if FPSCR.RN=0b10 then
   rnd ← bfp_ROUND_TO_INTEGER(0b010, src)
if FPSCR.RN=0b11 then
   rnd ← bfp_ROUND_TO_INTEGER(0b011, src)

result ← bfp64_CONVERT_FROM_BFP(rnd)

if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
if xx_flag=1     then SetFX(FPSCR.XX)

vex_flag ← FPSCR.VE & vxsnan_flag

if vex_flag=0 then do
   VSR[32×TX+T].dword[0] ← result
   VSR[32×TX+T].dword[1] ← 0x0000_0000_0000_0000
   FPSCR.FPRF ← fprf_CLASS_BFP64(result)
   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].

src is rounded to an integer using the rounding mode specified by RN. If src is a Signalling NaN, the result is the corresponding Quiet NaN, and VXSNAN is set to 1.

The result is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are set to 0.

FPRF is set to the class and sign of the result. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.

If a trap-enabled invalid operation exception occurs, VSR[XT] and FPRF are not modified, and FR and FI are set to 0.

Special Registers Altered:

Register	*Field(s)*
FPSCR	FPRF FR FI FX XX
	VXSNAN

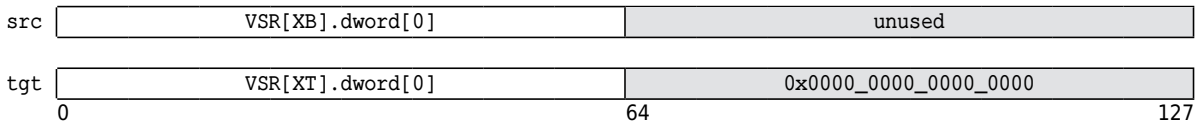
Programming Note

This instruction can be used to operate on a single-precision source operand.

Programming Note

Previous versions of the architecture allowed the contents of doubleword 1 of the result register to be undefined. However, all processors that support this instruction write 0s into doubleword 1 of the result register, as is required by this version of the architecture.

VSR Data Layout for xsrdpic



VSX Scalar Round to Double-Precision Integer using round toward -Infinity XX2-form

xsrdpim XT,XB

60	T	///	B	121	BX	TX
0	6	11	16	21	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

reset_xflags()

src    ← bfp_CONVERT_FROM_BFP64(VSR[VRB+32].dword[0])
rnd    ← bfp_ROUND_TO_INTEGER(0b011, src)
result ← bfp64_CONVERT_FROM_BFP(rnd)

if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)

vex_flag ← FPSCR.VE & vxsnan_flag

if vex_flag=0 then do
   VSR[32×TX+T].dword[0] ← result
   VSR[32×TX+T].dword[1] ← 0x0000_0000_0000_0000
   FPSCR.FPRF ← fprf_CLASS_BFP64(result)
end
FPSCR.FR ← 0b0
FPSCR.FI ← 0b0
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

Let src be the double-precision floating-point value in doubleword element 0 of VSR[XB].

src is rounded to an integer using the rounding mode Round toward −Infinity. If src is a Signalling NaN, the result is the corresponding Quiet NaN, and VXSNAN is set to 1.

The result is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are set to 0.

FPRF is set to the class and sign of the result. FR is set to 0. FI is set to 0.

If a trap-enabled invalid operation exception occurs, VSR[XT] and FPRF are not modified, and FR and FI are set to 0.

Special Registers Altered:

Register	*Field(s)*
FPSCR	FPRF FX VXSNAN
FPSCR	FR (set to 0)
FPSCR	FI (set to 0)

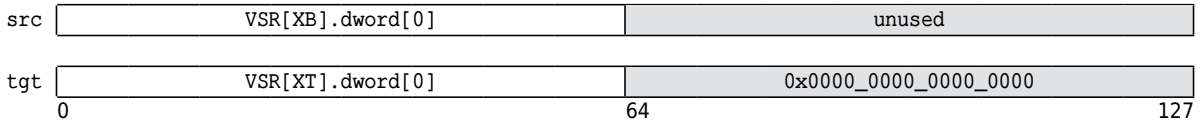
Programming Note

This instruction can be used to operate on a single-precision source operand.

Programming Note

Previous versions of the architecture allowed the contents of doubleword 1 of the result register to be undefined. However, all processors that support this instruction write 0s into doubleword 1 of the result register, as is required by this version of the architecture.

VSR Data Layout for xsrdpim



VSX Scalar Round to Double-Precision Integer using round toward +Infinity XX2-form

xsrdpip XT,XB

60	T	///	B	105	BX	TX
0	6	11	16	21	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

reset_xflags()

src    ← bfp_CONVERT_FROM_BFP64(VSR[VRB+32].dword[0])
rnd    ← bfp_ROUND_TO_INTEGER(0b010, src)
result ← bfp64_CONVERT_FROM_BFP(rnd)

if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)

vex_flag ← FPSCR.VE & vxsnan_flag

if vex_flag=0 then do
   VSR[32×TX+T].dword[0] ← result
   VSR[32×TX+T].dword[1] ← 0x0000_0000_0000_0000
   FPSCR.FPRF ← fprf_CLASS_BFP64(result)
end
FPSCR.FR ← 0b0
FPSCR.FI ← 0b0
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

Let src be the double-precision floating-point value in doubleword element 0 of VSR[XB].

src is rounded to an integer using the rounding mode Round toward +Infinity. If src is a Signalling NaN, the result is the corresponding Quiet NaN, and VXSNAN is set to 1.

The result is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are set to 0.

FPRF is set to the class and sign of the result. FR is set to 0. FI is set to 0.

If a trap-enabled invalid operation exception occurs, VSR[XT] and FPRF are not modified, and FR and FI are set to 0.

Special Registers Altered:

Register	*Field(s)*
FPSCR	FPRF FX VXSNAN
FPSCR	FR (set to 0)
FPSCR	FI (set to 0)

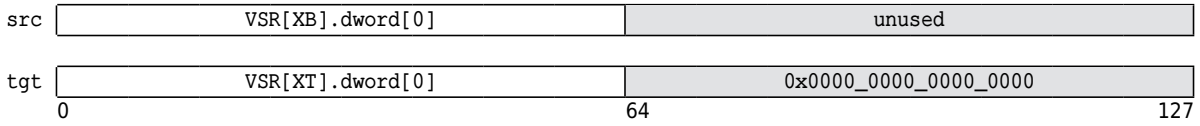
Programming Note

This instruction can be used to operate on a single-precision source operand.

Programming Note

Previous versions of the architecture allowed the contents of doubleword 1 of the result register to be undefined. However, all processors that support this instruction write 0s into doubleword 1 of the result register, as is required by this version of the architecture.

VSR Data Layout for xsrdpip



VSX Scalar Round to Double-Precision Integer using round toward Zero XX2-form

xsrdpiz XT,XB

60	T	///	B	89	BX	TX
0	6	11	16	21	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

reset_xflags()

src    ← bfp_CONVERT_FROM_BFP64(VSR[VRB+32].dword[0])
rnd    ← bfp_ROUND_TO_INTEGER(0b001, src)
result ← bfp64_CONVERT_FROM_BFP(rnd)

if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)

vex_flag ← FPSCR.VE & vxsnan_flag

if vex_flag=0 then do
   VSR[32×TX+T].dword[0] ← result
   VSR[32×TX+T].dword[1] ← 0x0000_0000_0000_0000
   FPSCR.FPRF ← fprf_CLASS_BFP64(result)
end
FPSCR.FR ← 0b0
FPSCR.FI ← 0b0
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

Let src be the double-precision floating-point value in doubleword element 0 of VSR[XB].

src is rounded to an integer using the rounding mode Round toward Zero. If src is a Signalling NaN, the result is the corresponding Quiet NaN, and VXSNAN is set to 1.

The result is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are set to 0.

FPRF is set to the class and sign of the result. FR is set to 0. FI is set to 0.

If a trap-enabled invalid operation exception occurs, VSR[XT] and FPRF are not modified, and FR and FI are set to 0.

Special Registers Altered:

Register	*Field(s)*
FPSCR	FPRF FX VXSNAN
FPSCR	FR (set to 0)
FPSCR	FI (set to 0)

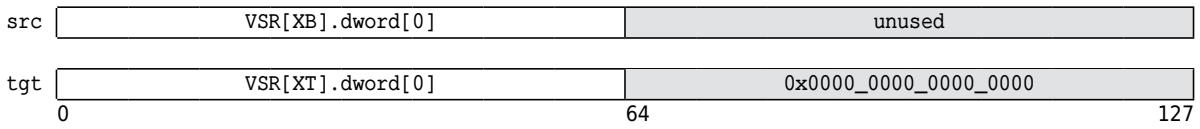
Programming Note

This instruction can be used to operate on a single-precision source operand.

Programming Note

Previous versions of the architecture allowed the contents of doubleword 1 of the result register to be undefined. However, all processors that support this instruction write 0s into doubleword 1 of the result register, as is required by this version of the architecture.

VSR Data Layout for xsrdpiz



VSX Scalar Round to Quad-Precision Integer [with Inexact] Z23-form

xsrqpi	R,VRT,VRB,RMC	(EX=0)
xsrqpix	R,VRT,VRB,RMC	(EX=1)

63	VRT	///	R	VRB	RMC	5	EX
0	6	11	15	16	21	23	31

```
if MSR.VSX=0 then VSX_Unavailable()

reset_xflags()

if R=0 then do
   if RMC=0b00 then           // Round to Nearest Away
      rmode ← 0b100
   if RMC=0b11 then do
      if FPSCR.RN=0b00 then   // Round to Nearest Even
         rmode ← 0b000
      if FPSCR.RN=0b01 then   // Round towards Zero
         rmode ← 0b001
      if FPSCR.RN=0b10 then   // Round towards +Infinity
         rmode ← 0b010
      if FPSCR.RN=0b11 then   // Round towards -Infinity
         rmode ← 0b011
   end
end
else do // R=1
   if RMC=0b00 then           // Round to Nearest Even
      rmode ← 0b000
   if RMC=0b01 then           // Round towards Zero
      rmode ← 0b001
   if RMC=0b10 then           // Round towards +Infinity
      rmode ← 0b010
   if RMC=0b11 then           // Round towards -Infinity
      rmode ← 0b011
end

src ← bfp_CONVERT_FROM_BFP128(VSR[VRB+32])

if src.class.SNaN then do
   result ← bfp128_CONVERT_FROM_BFP(bfp_QUIET(src))
   vxsnan_flag ← 1
end
else if src.class.QNaN     |
        src.class.Infinity |
        src.class.Zero  then
   result ← bfp128_CONVERT_FROM_BFP(src)
else do
   rnd    ← bfp_ROUND_TO_INTEGER(rmode, src)
   result ← bfp128_CONVERT_FROM_BFP(rnd)
end

if vxsnan_flag=1  then SetFX(FPSCR.VXSNAN)
if xx_flag & EX then SetFX(FPSCR.XX)

vex_flag  ← FPSCR.VE & vxsnan_flag

if vex_flag=0 then do
   VSR[VRT+32] ← result
   FPSCR.FPRF  ← fprf_CLASS_BFP128(result)
end
FPSCR.FR ← EX & (vxsnan_flag=0) & inc_flag
FPSCR.FI ← EX & (vxsnan_flag=0) & xx_flag
```

Let R and RMC specify the rounding mode as follows.

R	RMC	FPSCR.RN	Rounding Mode
0	00	–	Round to Nearest Away
0	01	–	reserved
0	10	–	reserved
0	11	00	Round to Nearest Even
0	11	01	Round towards Zero
0	11	10	Round towards +Infinity
0	11	11	Round towards −Infinity
1	00	–	Round to Nearest Even
1	01	–	Round towards Zero
1	10	–	Round towards +Infinity
1	11	–	Round towards −Infinity

Let src be the floating-point value in VSR[VRB+32] represented in quad-precision format.

If src is a Signalling NaN, an Invalid Operation exception occurs, VXSNAN is set to 1, and the result is the Quiet NaN corresponding to the Signalling NaN.

Otherwise, if src is a Quiet NaN, an Infinity, or a Zero, then the result is src.

Otherwise, src is rounded to an integer using the rounding mode rmode.

The result is placed into VSR[VRT+32] in quad-precision format.

FPRF is set to the class and sign of the result.

For ***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.

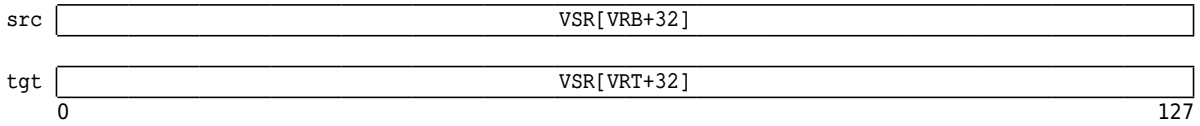
If a trap-disabled Invalid Operation exception occurs, FPRF is set to an undefined value.

If a trap-enabled Invalid Operation exception occurs, VSR[VRT+32] and FPRF are not modified.

Special Registers Altered:

Register	*Field(s)*	
FPSCR	FPRF VXSNAN FX	
FPSCR	FR	(if ***xsrqpi***)
(set to 0)		
FPSCR	FI	(if ***xsrqpi***)
(set to 0)		
FPSCR	FR FI XX	(if ***xsrqpix***)

VSR Data Layout for xsrqpi[x]



7.6.2.8.2 VSX Vector BFP Round to Integral Instructions

VSX Vector Round to Double-Precision Integer using round to Nearest Away XX2-form

xvrdpi XT,XB

60	T	///	B	201	BX	TX
0	6	11	16	21	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

ex_flag ← 0b0

do i = 0 to 1
   reset_xflags()

   src ← bfp_CONVERT_FROM_BFP64(VSR[32×BX+B].dword[i])
   rnd ← bfp_ROUND_TO_INTEGER(0b100, src)

   vresult.dword[i] ← bfp64_CONVERT_FROM_BFP(rnd)

   if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)

   ex_flag ← ex_flag | (FPSCR.VE & vxsnan_flag)
end

if ex_flag=0 then VSR[32×TX+T] ← vresult
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

For each integer value i from 0 to 1, do the following.

Let src be the double-precision floating-point operand in doubleword element i of VSR[XB].

src is rounded to an integer using the rounding mode Round to Nearest Away. If src is a Signalling NaN, the result is the corresponding Quiet NaN, and VXSNAN is set to 1.

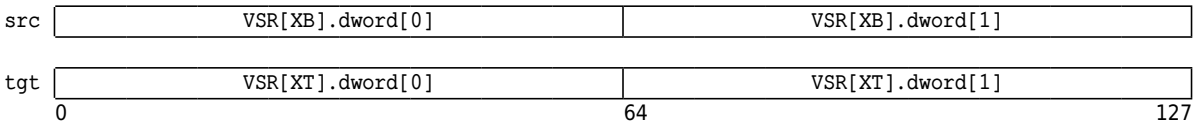
The result is placed into doubleword element i of VSR[XT] in double-precision format.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered:

Register	*Field(s)*
FPSCR	FX VXSNAN

VSR Data Layout for xvrdpi



VSX Vector Round to Double-Precision Integer Exact using Current rounding mode XX2-form

xvrdpic XT,XB

60	T	///	B	235	BX	TX
0	6	11	16	21	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

ex_flag ← 0b0

do i = 0 to 1
   reset_xflags()

   src ← bfp_CONVERT_FROM_BFP64(VSR[32×BX+B].dword[i])

   if FPSCR.RN=0b00 then
      rnd ← bfp_ROUND_TO_INTEGER(0b000, src)
   if FPSCR.RN=0b01 then
      rnd ← bfp_ROUND_TO_INTEGER(0b001, src)
   if FPSCR.RN=0b10 then
      rnd ← bfp_ROUND_TO_INTEGER(0b010, src)
   if FPSCR.RN=0b11 then
      rnd ← bfp_ROUND_TO_INTEGER(0b011, src)

   vresult.dword[i] ← bfp64_CONVERT_FROM_BFP(rnd)

   if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
   if xx_flag=1 then SetFX(FPSCR.XX)

   ex_flag ← ex_flag | (FPSCR.VE & vxsnan_flag)
   ex_flag ← ex_flag | (FPSCR.XE & xx_flag)
end

if ex_flag=0 then VSR[32×TX+T] ← vresult
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

For each integer value i from 0 to 1, do the following.

Let src be the double-precision floating-point operand in doubleword element i of VSR[XB].

src is rounded to an integer using the rounding mode specified by RN. If src is a Signalling NaN, the result is the corresponding Quiet NaN, and VXSNAN is set to 1.

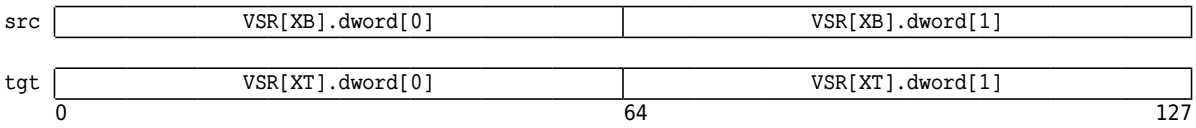
The result is placed into doubleword element i of VSR[XT] in double-precision format.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered:

Register	*Field(s)*
FPSCR	FX XX VXSNAN

VSR Data Layout for xvrdpic



VSX Vector Round to Double-Precision Integer using round toward -Infinity XX2-form

xvrdpim XT,XB

60	T	///	B	249	BX	TX
0	6	11	16	21	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

ex_flag ← 0b0

do i = 0 to 1
   reset_xflags()

   src ← bfp_CONVERT_FROM_BFP64(VSR[32×BX+B].dword[i])
   rnd ← bfp_ROUND_TO_INTEGER(0b011, src)

   vresult.dword[i] ← bfp64_CONVERT_FROM_BFP(rnd)

   if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)

   ex_flag ← ex_flag | (FPSCR.VE & vxsnan_flag)
end

if ex_flag=0 then VSR[32×TX+T] ← vresult
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

For each integer value i from 0 to 1, do the following.

Let src be the double-precision floating-point operand in doubleword element i of VSR[XB].

src is rounded to an integer using the rounding mode Round toward −Infinity. If src is a Signalling NaN, the result is the corresponding Quiet NaN, and VXSNAN is set to 1.

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:

Register	*Field(s)*
FPSCR	FX VXSNAN

VSX Vector Round to Double-Precision Integer using round toward +Infinity XX2-form

xvrdpip XT,XB

60	T	///	B	233	BX	TX
0	6	11	16	21	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

ex_flag ← 0b0

do i = 0 to 1
   reset_xflags()

   src ← bfp_CONVERT_FROM_BFP64(VSR[32×BX+B].dword[i])
   rnd ← bfp_ROUND_TO_INTEGER(0b010, src)

   vresult.dword[i] ← bfp64_CONVERT_FROM_BFP(rnd)

   if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)

   ex_flag ← ex_flag | (FPSCR.VE & vxsnan_flag)
end

if ex_flag=0 then VSR[32×TX+T] ← vresult
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

For each integer value i from 0 to 1, do the following.

Let src be the double-precision floating-point operand in doubleword element i of VSR[XB].

src is rounded to an integer using the rounding mode Round toward +Infinity. If src is a Signalling NaN, the result is the corresponding Quiet NaN, and VXSNAN is set to 1.

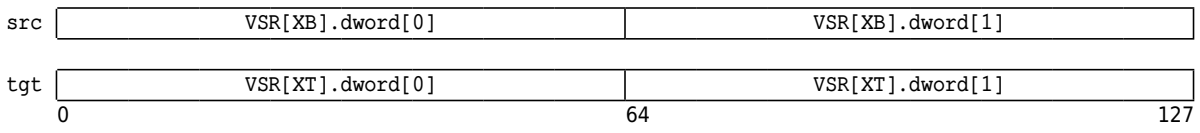
The result is placed into doubleword element i of VSR[XT] in double-precision format.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered:

Register	*Field(s)*
FPSCR	FX VXSNAN

VSR Data Layout for xvrdpim & xvrdpip



VSX Vector Round to Double-Precision Integer using round toward Zero XX2-form

xvrdpiz XT,XB

60	T	///	B	217	BX	TX
0	6	11	16	21	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

ex_flag ← 0b0

do i = 0 to 1
   reset_xflags()

   src ← bfp_CONVERT_FROM_BFP64(VSR[32×BX+B].dword[i])
   rnd ← bfp_ROUND_TO_INTEGER(0b001, src)

   vresult.dword[i] ← bfp64_CONVERT_FROM_BFP(rnd)

   if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)

   ex_flag ← ex_flag | (FPSCR.VE & vxsnan_flag)
end

if ex_flag=0 then VSR[32×TX+T] ← vresult
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

For each integer value i from 0 to 1, do the following.

Let src be the double-precision floating-point operand in doubleword element i of VSR[XB].

src is rounded to an integer using the rounding mode Round toward Zero. If src is a Signalling NaN, the result is the corresponding Quiet NaN, and VXSNAN is set to 1.

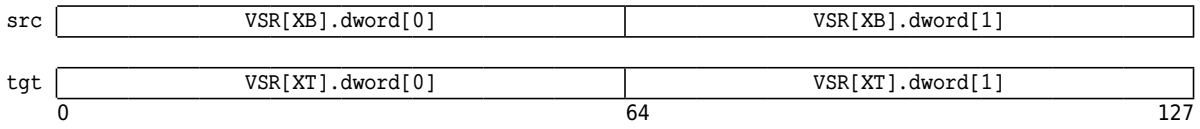
The result is placed into doubleword element i of VSR[XT] in double-precision format.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered:

Register	*Field(s)*
FPSCR	FX VXSNAN

VSR Data Layout for xvrdpiz



VSX Vector Round to Single-Precision Integer using round to Nearest Away XX2-form

xvrspi XT,XB

60	T	///	B	137	BX	TX
0	6	11	16	21	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

ex_flag ← 0b0

do i = 0 to 3
   reset_xflags()
 <
   src ← bfp_CONVERT_FROM_BFP32(VSR[32×BX+B].word[i])
   rnd ← bfp_ROUND_TO_INTEGER(0b100, src)

   vresult.word[i] ← bfp32_CONVERT_FROM_BFP(rnd)

   if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)

   ex_flag ← ex_flag | (FPSCR.VE & vxsnan_flag)
end

if ex_flag=0 then VSR[32×TX+T] ← vresult
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

For each integer value i from 0 to 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 to Nearest Away. If src is a Signalling NaN, the result is the corresponding Quiet NaN, and VXSNAN is set to 1.

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:

Register	*Field(s)*
FPSCR	FX VXSNAN

VSR Data Layout for xvrspi

VSX Vector Round to Single-Precision Integer Exact using Current rounding mode XX2-form

xvrspic XT,XB

60	T	///	B	171	BX	TX
0	6	11	16	21	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

ex_flag ← 0b0

do i = 0 to 3
   reset_xflags()

   src ← bfp_CONVERT_FROM_BFP32(VSR[32×BX+B].word[i])
   if FPSCR.RN=0b00 then
      rnd ← bfp_ROUND_TO_INTEGER(0b000, src)
   if FPSCR.RN=0b01 then
      rnd ← bfp_ROUND_TO_INTEGER(0b001, src)
   if FPSCR.RN=0b10 then
      rnd ← bfp_ROUND_TO_INTEGER(0b010, src)
   if FPSCR.RN=0b11 then
      rnd ← bfp_ROUND_TO_INTEGER(0b011, src)

   vresult.word[i] ← bfp32_CONVERT_FROM_BFP(rnd)

   if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
   if xx_flag=1     then SetFX(FPSCR.XX)

   ex_flag ← ex_flag | (FPSCR.VE & vxsnan_flag)
                     | (FPSCR.XE & xx_flag)
end

if ex_flag=0 then VSR[32×TX+T] ← vresult
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

For each integer value i from 0 to 3, do the following.

Let src be the 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. If src is a Signalling NaN, the result is the corresponding Quiet NaN, and VXSNAN is set to 1.

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:

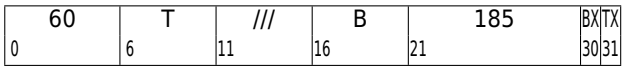
Register	*Field(s)*
FPSCR	FX XX VXSNAN

VSR Data Layout for xvrspic

src	VSR[XB].word[0]	VSR[XB].word[1]	VSR[XB].word[2]	VSR[XB].word[3]
tgt	VSR[XT].word[0]	VSR[XT].word[1]	VSR[XT].word[2]	VSR[XT].word[3]
	0	32	64	96 127

VSX Vector Round to Single-Precision Integer using round toward -Infinity XX2-form

xvrspim XT,XB




```
if MSR.VSX=0 then VSX_Unavailable()

ex_flag ← 0b0

do i = 0 to 3
   reset_xflags()

   src ← bfp_CONVERT_FROM_BFP32(VSR[32×BX+B].word[i])
   rnd ← bfp_ROUND_TO_INTEGER(0b011, src)

   vresult.word[i] ← bfp32_CONVERT_FROM_BFP(rnd)

   if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)

   ex_flag ← ex_flag | (FPSCR.VE & vxsnan_flag)
end

if ex_flag=0 then VSR[32×TX+T] ← vresult
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

For each integer value i from 0 to 3, do the following.

Let src be the single-precision floating-point operand in word element i of VSR[XB].

src is rounded to an integer using the rounding mode Round toward −Infinity. If src is a Signalling NaN, the result is the corresponding Quiet NaN, and VXSNAN is set to 1.

The result is placed into word element i of VSR[XT] in single-precision format.

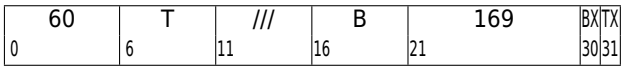
If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered:

Register	*Field(s)*
FPSCR	FX VXSNAN

VSX Vector Round to Single-Precision Integer using round toward +Infinity XX2-form

xvrspip XT,XB




```
if MSR.VSX=0 then VSX_Unavailable()

ex_flag ← 0b0

do i = 0 to 3
   reset_xflags()

   src ← bfp_CONVERT_FROM_BFP32(VSR[32×BX+B].word[i])
   rnd ← bfp_ROUND_TO_INTEGER(0b010, src)

   vresult.word[i] ← bfp32_CONVERT_FROM_BFP(rnd)

   if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)

   ex_flag ← ex_flag | (FPSCR.VE & vxsnan_flag)
end

if ex_flag=0 then VSR[32×TX+T] ← vresult
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

For each integer value i from 0 to 3, do the following.

Let src be the single-precision floating-point operand in word element i of VSR[XB].

src is rounded to an integer using the rounding mode Round toward +Infinity. If src is a Signalling NaN, the result is the corresponding Quiet NaN, and VXSNAN is set to 1.

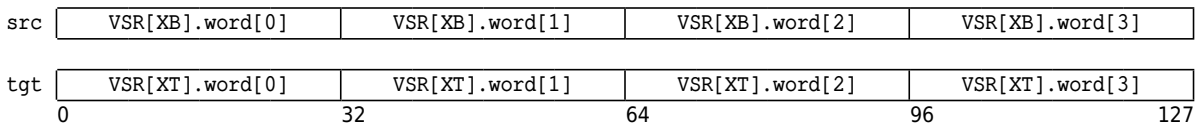
The result is placed into word element i of VSR[XT] in single-precision format.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered:

Register	*Field(s)*
FPSCR	FX VXSNAN

VSR Data Layout for xvrspim & xvrspip



VSX Vector Round to Single-Precision Integer using round toward Zero XX2-form

xvrspiz XT,XB

60	T	///	B	153	BX	TX
0	6	11	16	21	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

ex_flag ← 0b0

do i = 0 to 3
   reset_xflags()

   src ← bfp_CONVERT_FROM_BFP32(VSR[32×BX+B].word[i])
   rnd ← bfp_ROUND_TO_INTEGER(0b001, src)

   vresult.word[i] ← bfp32_CONVERT_FROM_BFP(rnd)

   if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)

   ex_flag ← ex_flag | (FPSCR.VE & vxsnan_flag)
end

if ex_flag=0 then VSR[32×TX+T] ← vresult
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

For each integer value i from 0 to 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. If src is a Signalling NaN, the result is the corresponding Quiet NaN, and VXSNAN is set to 1.

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:

Register	*Field(s)*
FPSCR	FX VXSNAN

VSR Data Layout for xvrspiz

7.6.2.9 VSX Binary Floating-Point Convert To Integer Instructions

7.6.2.9.1 VSX Scalar BFP Convert To Integer Instructions

VSX Scalar Convert with round to zero Double-Precision to Signed Doubleword format XX2-form

xscvdpsxds XT,XB

60	T	///	B	344	BX	TX
0	6	11	16	21	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

reset_xflags()

src    ← bfp_CONVERT_FROM_BFP64(VSR[32×BX+B].dword[0])
rnd    ← bfp_ROUND_TO_INTEGER(0b001, src)
result ← si64_CONVERT_FROM_BFP(rnd)

if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
if vxcvi_flag=1 then SetFX(FPSCR.VXCVI)
if xx_flag=1 then SetFX(FPSCR.XX)

vx_flag  ← vxsnan_flag | vxcvi_flag
vex_flag ← FPSCR.VE & vx_flag

if vex_flag=0 then do
   VSR[32×TX+T].dword[1] ← result
   VSR[32×TX+T].dword[2] ← 0x0000_0000_0000_0000
   FPSCR.FPRF ← 0bUUUUU
   FPSCR.FR   ← inc_flag
   FPSCR.FI   ← xx_flag
end
else do
   FPSCR.FR   ← 0b0
   FPSCR.FI   ← 0b0
end
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

Let src be the double-precision floating-point value in doubleword element 0 of VSR[XB].

If src is a NaN, the result is the value 0x8000_0000_0000_0000 and VXCVI is set to 1. If src is an 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 $2^{63} - 1$, the result is 0x7FFF_FFFF_FFFF_FFFF and VXCVI is set to 1.

Otherwise, if the rounded value is less than -2^{63}, the result is 0x8000_0000_0000_0000 and VXCVI is set to 1.

Otherwise, the result is the rounded value converted to 64-bit signed-integer format, and if the result is inexact (i.e., not equal to src), XX is set to 1.

If a trap-enabled invalid operation exception occurs,

- VSR[XT] and FPRF are not modified
- FR and FI are set to 0.

Otherwise,

- The result is placed into doubleword element 0 of VSR[XT]. The contents of doubleword element 1 of VSR[XT] are set to 0.
- FPRF is set to an undefined value.
- FR is set to indicate if the result was incremented when rounded.
- FI is set to indicate the result is inexact.

See Table 7.74.

Special Registers Altered:

Register	*Field(s)*
FPSCR	FPRF (undefined)
FPSCR	FR FI FX XX VXSNAN VXCVI

Programming Note

This instruction can be used to operate on a single-precision source operand.

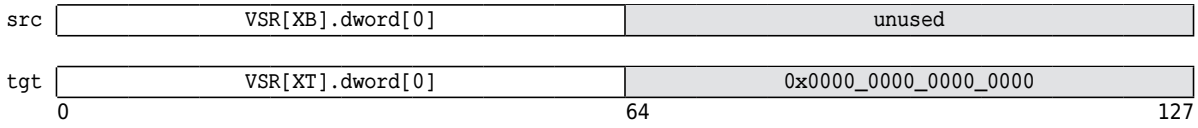
Programming Note

xscvdpsxds rounds using Round towards Zero rounding mode. For other rounding modes, software must use a *Round to Double-Precision Integer* instruction that corresponds to the desired rounding mode, including ***xsrdpic*** which uses the rounding mode specified by RN.

Programming Note

Previous versions of the architecture allowed the contents of doubleword 1 of the result register to be undefined. However, all processors that support this instruction write 0s into doubleword 1 of the result register, as is required by this version of the architecture.

VSR Data Layout for xscvdpsxds



	FPSCR.VE	FPSCR.XE	Inexact? (trunc(src) ≠ src)	Returned Results and Status Setting
src ≤ Nmin−1	0	–	–	T(Nmin), fr(0), fi(0), fx(VXCVI)
	1	–	–	fr(0), fi(0), fx(VXCVI), error()
Nmin−1 < src < Nmin	–	0	yes	T(Nmin), fr(0), fi(1), fx(XX)
		1	yes	T(Nmin), fr(0), fi(1), fx(XX), error()
src = Nmin	–	–	no	T(Nmin), fr(0), fi(0)
Nmin < src < Nmax	–	–	no	T(f2i(trunc(src))), fr(0), fi(0)
		0	yes	T(f2i(trunc(src))), fr(0), fi(1), fx(XX)
		1	yes	T(f2i(trunc(src))), fr(0), fi(1), fx(XX), error()
src = Nmax	–	–	no	T(Nmax), fr(0), fi(0) Note: This case cannot occur as Nmax is not representable in DP format but is included here for completeness.
Nmax < src < Nmax+1	–	0	yes	T(Nmax), fr(0), fi(1), fx(XX)
		1	yes	T(Nmax), fr(0), fi(1), fx(XX), error()
src ≥ Nmax+1	0	–	–	T(Nmax), fr(0), fi(0), fx(VXCVI)
	1	–	–	fr(0), fi(0), fx(VXCVI), error()
src is a QNaN	0	–	–	T(Nmin), fr(0), fi(0), fx(VXCVI)
	1	–	–	fr(0), fi(0), fx(VXCVI), error()
src is a SNaN	0	–	–	T(Nmin), fr(0), fi(0), fx(VXCVI), fx(VXSNAN)
	1	–	–	fr(0), fi(0), fx(VXCVI), fx(VXSNAN), error()

Explanation:

error()	The system error handler is invoked for the trap-enabled exception if MSR.FE0 and MSR.FE1 bits are set to any mode other than the ignore-exception mode.
f2i(x)	The double-precision floating-point integer value x is converted to 64-bit signed integer format.
fi(x)	FPSCR.FI is set to the value x.
fr(x)	FPSCR.FR is set to the value x.
fx(x)	FPSCR.FX is set to 1 if FPSCR.x=0. FPSCR.x is set to 1.
Nmin	The smallest signed integer doubleword value, -2^{63} (0x8000_0000_0000_0000).
Nmax	The largest signed integer doubleword value, $2^{63}-1$ (0x7FFF_FFFF_FFFF_FFFF).
src	The double-precision floating-point value in doubleword element 0 of VSR[XB].
trunc(x)	The double-precision floating-point value x is truncated to a floating-point integer.
T(x)	The signed integer doubleword value x is placed in doubleword element 0 of VSR[XT]. The contents of doubleword element 1 of VSR[XT] are set to 0.

Table 7.74: Actions for xscvdpsxds

VSX Scalar Convert with round to zero Double-Precision to Signed Word format XX2-form

xscvdpsxws XT,XB

60	T	///	B	88	BX	TX
0	6	11	16	21	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

reset_xflags()

src    ← bfp_CONVERT_FROM_BFP64(VSR[32×BX+B].dword[0])
rnd    ← bfp_ROUND_TO_INTEGER(0b001, src)
result ← si32_CONVERT_FROM_BFP(rnd)

if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
if vxcvi_flag=1 then SetFX(FPSCR.VXCVI)
if xx_flag=1 then SetFX(FPSCR.XX)

vx_flag  ← vxsnan_flag | vxcvi_flag
vex_flag ← FPSCR.VE & vx_flag

if vex_flag=0 then do
   VSR[32×TX+T].word[0] ← result
   VSR[32×TX+T].word[1] ← result
   VSR[32×TX+T].word[2] ← 0x0000_0000
   VSR[32×TX+T].word[3] ← 0x0000_0000
   FPSCR.FPRF ← 0bUUUUU
   FPSCR.FR   ← inc_flag
   FPSCR.FI   ← xx_flag
end
else do
   FPSCR.FR   ← 0b0
   FPSCR.FI   ← 0b0
end
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

Let src be the double-precision floating-point value in doubleword element 0 of VSR[XB].

If src is a NaN, the result is the value 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.

Otherwise,

- The result is placed into word elements 0 and 1 of VSR[XT]. The contents of word elements 2 and 3 of VSR[XT] are set to 0.
- FPRF is set to an undefined value.
- FR is set to indicate if the result was incremented when rounded.
- FI is set to indicate the result is inexact.

See Table 7.75.

Special Registers Altered:

Register	*Field(s)*
FPSCR	FPRF (undefined)
FPSCR	FR FI FX XX VXSNAN VXCVI

Programming Note

Previous versions of the architecture allowed the contents of word 0 of the result register to be undefined. However, all processors that support this instruction write the result into words 0 and 1 of the result register, as is required by this version of the architecture.

Programming Note

Previous versions of the architecture allowed the contents of doubleword 1 of the result register to be undefined. However, all processors that support this instruction write 0s into doubleword 1 of the result register, as is required by this version of the architecture.

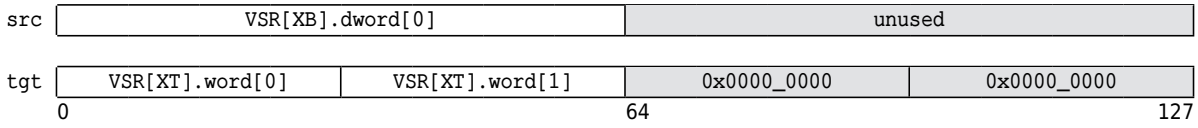
Programming Note

This instruction can be used to operate on a single-precision source operand.

Programming Note

xscvdpsxws rounds using Round towards Zero rounding mode. For other rounding modes, software must use a *Round to Double-Precision Integer* instruction that corresponds to the desired rounding mode, including ***xsrdpic*** which uses the rounding mode specified by RN.

VSR Data Layout for xscvdpsxws



	FPSCR.VE	FPSCR.XE	Inexact? (trunc(src) ≠ src)	Returned Results and Status Setting
src ≤ Nmin−1	0	–	–	T(Nmin), fr(0), fi(0), fx(VXCVI)
	1	–	–	fr(0), fi(0), fx(VXCVI), error()
Nmin−1 < src < Nmin	–	0	yes	T(Nmin), fr(0), fi(1), fx(XX)
		1	yes	T(Nmin), fr(0), fi(1), fx(XX), error()
src = Nmin	–	–	no	T(Nmin), fr(0), fi(0)
Nmin < src < Nmax	–	–	no	T(f2i(trunc(src))), fr(0), fi(0)
		0	yes	T(f2i(trunc(src))), fr(0), fi(1), fx(XX)
		1	yes	T(f2i(trunc(src))), fr(0), fi(1), fx(XX), error()
src = Nmax	–	–	no	T(Nmax), fr(0), fi(0)
Nmax < src < Nmax+1	–	0	yes	T(Nmax), fr(0), fi(1), fx(XX)
		1	yes	T(Nmax), fr(0), fi(1), fx(XX), error()
src ≥ Nmax+1	0	–	–	T(Nmax), fr(0), fi(0), fx(VXCVI)
	1	–	–	fr(0), fi(0), fx(VXCVI), error()
src is a QNaN	0	–	–	T(Nmin), fr(0), fi(0), fx(VXCVI)
	1	–	–	fr(0), fi(0), fx(VXCVI), error()
src is a SNaN	0	–	–	T(Nmin), fr(0), fi(0), fx(VXCVI), fx(VXSNAN)
	1	–	–	fr(0), fi(0), fx(VXCVI), fx(VXSNAN), error()

Explanation:

error()	The system error handler is invoked for the trap-enabled exception if MSR.FE0 and MSR.FE1 bits are set to any mode other than the ignore-exception mode.
f2i(x)	The double-precision floating-point integer value x is converted to 32-bit signed integer format.
fi(x)	FPSCR.FI is set to the value x.
fr(x)	FPSCR.FR is set to the value x.
fx(x)	FPSCR.FX is set to 1 if FPSCR.x=0. FPSCR.x is set to 1.
Nmin	The smallest signed integer 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].
trunc(x)	The double-precision floating-point value x is truncated to a floating-point integer.
T(x)	The signed integer word value x is placed in word elements 0 and 1 of VSR[XT]. The contents of word elements 2 and 3 of VSR[XT] are set to 0.

Table 7.75: Actions for xscvdpsxws

VSX Scalar Convert with round to zero Double-Precision to Unsigned Doubleword format XX2-form

xscvdpuxds XT,XB

60	T	///	B	328	BX	TX
0	6	11	16	21	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

reset_xflags()

src    ← bfp_CONVERT_FROM_BFP64(VSR[32×BX+B].dword[0])
rnd    ← bfp_ROUND_TO_INTEGER(0b001, src)
result ← ui64_CONVERT_FROM_BFP(rnd)

if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
if vxcvi_flag=1 then SetFX(FPSCR.VXCVI)
if xx_flag=1 then SetFX(FPSCR.XX)

vx_flag  ← vxsnan_flag | vxcvi_flag
vex_flag ← FPSCR.VE & vx_flag

if vex_flag=0 then do
   VSR[32×TX+T].dword[1] ← result
   VSR[32×TX+T].dword[2] ← 0x0000_0000_0000_0000
   FPSCR.FPRF ← 0bUUUUU
   FPSCR.FR   ← inc_flag
   FPSCR.FI   ← xx_flag
end
else do
   FPSCR.FR   ← 0b0
   FPSCR.FI   ← 0b0
end
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

Let src be the double-precision floating-point value in doubleword element 0 of VSR[XB].

If src is a NaN, the result is the value 0x0000_0000_0000_0000 and VXCVI is set to 1. If src is an SNaN, VXSNAN is also set to 1.

Otherwise, src is rounded to a floating-point integer using the rounding mode Round Toward Zero.

If the rounded value is greater than $2^{64} - 1$, the result is 0xFFFF_FFFF_FFFF_FFFF and VXCVI is set to 1.

Otherwise, if the rounded value is less than 0, the result is 0x0000_0000_0000_0000 and VXCVI is set to 1.

Otherwise, the result is the rounded value converted to 64-bit unsigned-integer format, and if the result is inexact (i.e., not equal to src), XX is set to 1.

If a trap-enabled invalid operation exception occurs,

- 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 set to 0.
- FPRF is set to an undefined value.
- FR is set to indicate if the result was incremented when rounded.
- FI is set to indicate the result is inexact.

See Table 7.76.

Special Registers Altered:

Register	*Field(s)*
FPSCR	FPRF (undefined)
FPSCR	FR FI FX XX VXSNAN VXCVI

Programming Note

Previous versions of the architecture allowed the contents of doubleword 1 of the result register to be undefined. However, all processors that support this instruction write 0s into doubleword 1 of the result register, as is required by this version of the architecture.

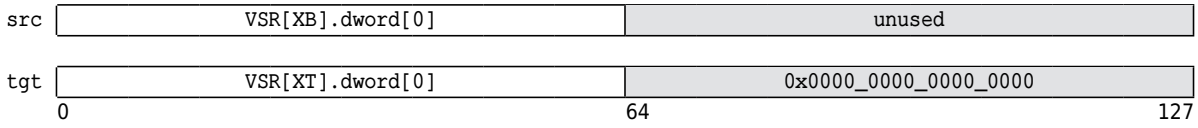
Programming Note

This instruction can be used to operate on a single-precision source operand.

Programming Note

xscvdpuxds rounds using Round towards Zero rounding mode. For other rounding modes, software must use a *Round to Double-Precision Integer* instruction that corresponds to the desired rounding mode, including ***xsrdpic*** which uses the rounding mode specified by RN.

VSR Data Layout for xscvdpuxds



	FPSCR.VE	FPSCR.XE	Inexact? (trunc(src) ≠ src)	Returned Results and Status Setting
src ≤ Nmin−1	0	–	–	T(Nmin), fr(0), fi(0), fx(VXCVI)
	1	–	–	fr(0), fi(0), fx(VXCVI), error()
Nmin−1 < src < Nmin	–	0	yes	T(Nmin), fr(0), fi(1), fx(XX)
		1	yes	T(Nmin), fr(0), fi(1), fx(XX), error()
src = Nmin	–	–	no	T(Nmin), fr(0), fi(0)
Nmin < src < Nmax	–	–	no	T(f2i(trunc(src))), fr(0), fi(0)
		0	yes	T(f2i(trunc(src))), fr(0), fi(1), fx(XX)
		1	yes	T(f2i(trunc(src))), fr(0), fi(1), fx(XX), error()
src = Nmax	–	–	no	T(Nmax), fr(0), fi(0) Note: This case cannot occur as Nmax is not representable in DP format but is included here for completeness.
Nmax < src < Nmax+1	–	0	yes	T(Nmax), fr(0), fi(1), fx(XX)
		1	yes	T(Nmax), fr(0), fi(1), fx(XX), error()
src ≥ Nmax+1	0	–	–	T(Nmax), fr(0), fi(0), fx(VXCVI)
	1	–	–	fr(0), fi(0), fx(VXCVI), error()
src is a QNaN	0	–	–	T(Nmin), fr(0), fi(0), fx(VXCVI)
	1	–	–	fr(0), fi(0), fx(VXCVI), error()
src is a SNaN	0	–	–	T(Nmin), fr(0), fi(0), fx(VXCVI), fx(VXSNAN)
	1	–	–	fr(0), fi(0), fx(VXCVI), fx(VXSNAN), error()

Explanation:

error()	The system error handler is invoked for the trap-enabled exception if MSR.FE0 and MSR.FE1 bits are set to any mode other than the ignore-exception mode.
f2i(x)	The double-precision floating-point integer value x is converted to 64-bit unsigned integer format.
fi(x)	FPSCR.FI is set to the value x.
fr(x)	FPSCR.FR is set to the value x.
fx(x)	FPSCR.FX is set to 1 if FPSCR.x=0. FPSCR.x is set to 1.
Nmin	The smallest unsigned integer doubleword value, 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 0 of VSR[XB].
trunc(x)	The double-precision floating-point value x is truncated to a floating-point integer.
T(x)	The signed integer doubleword value x is placed in doubleword element 0 of VSR[XT]. The contents of doubleword element 1 of VSR[XT] are set to 0.

Table 7.76: Actions for xscvdpuxds

VSX Scalar Convert with round to zero Double-Precision to Unsigned Word format XX2-form

xscvdpuxws XT,XB

60	T	///	B	72	BX	TX
0	6	11	16	21	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

reset_xflags()

src    ← bfp_CONVERT_FROM_BFP64(VSR[32×BX+B].dword[0])
rnd    ← bfp_ROUND_TO_INTEGER(0b001, src)
result ← ui32_CONVERT_FROM_BFP(rnd)

if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
if vxcvi_flag=1 then SetFX(FPSCR.VXCVI)
if xx_flag=1 then SetFX(FPSCR.XX)

vx_flag  ← vxsnan_flag | vxcvi_flag
vex_flag ← FPSCR.VE & vx_flag

if vex_flag=0 then do
   VSR[32×TX+T].word[0] ← result
   VSR[32×TX+T].word[1] ← result
   VSR[32×TX+T].word[2] ← 0x0000_0000
   VSR[32×TX+T].word[3] ← 0x0000_0000
   FPSCR.FPRF ← 0bUUUUU
   FPSCR.FR   ← inc_flag
   FPSCR.FI   ← xx_flag
end
else do
   FPSCR.FR   ← 0b0
   FPSCR.FI   ← 0b0
end
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

Let src be the double-precision floating-point value in doubleword element 0 of VSR[XB].

If src is a NaN, the result is the value 0x0000_0000 and VXCVI is set to 1. If src is an SNaN, VXSNAN is also set to 1.

Otherwise, src is rounded to a floating-point integer using the rounding mode Round Toward Zero.

If the rounded value is greater than $2^{32}-1$, the result is 0xFFFF_FFFF and VXCVI is set to 1.

Otherwise, if the rounded value is less than 0, the result is 0x0000_0000 and VXCVI is set to 1.

Otherwise, the result is the rounded value converted to 32-bit unsigned-integer format, and if the result is inexact (i.e., not equal to src), XX is set to 1.

If a trap-enabled invalid operation exception occurs,

- VSR[XT] and FPRF are not modified
- FR and FI are set to 0.

Otherwise,

- The result is placed into word elements 0 and 1 of VSR[XT]. The contents of word elements 2 and 3 of VSR[XT] are set to 0.
- FPRF is set to an undefined value.
- FR is set to indicate if the result was incremented when rounded.
- FI is set to indicate the result is inexact.

See Table 7.77.

Special Registers Altered:

Register	*Field(s)*
FPSCR	FPRF (undefined)
FPSCR	FR FI FX XX VXSNAN VXCVI

Programming Note

Previous versions of the architecture allowed the contents of word 0 of the result register to be undefined. However, all processors that support this instruction write the result into words 0 and 1 of the result register, as is required by this version of the architecture.

Programming Note

Previous versions of the architecture allowed the contents of doubleword 1 of the result register to be undefined. However, all processors that support this instruction write 0s into doubleword 1 of the result register, as is required by this version of the architecture.

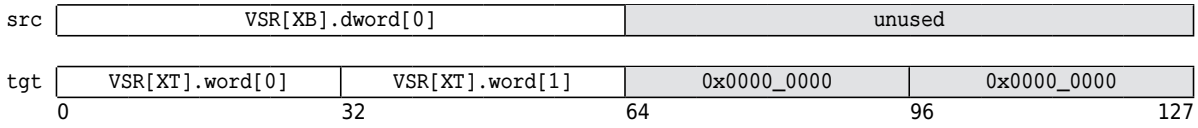
Programming Note

This instruction can be used to operate on a single-precision source operand.

Programming Note

xscvdpuxws rounds using Round towards Zero rounding mode. For other rounding modes, software must use a *Round to Double-Precision Integer* instruction that corresponds to the desired rounding mode, including ***xsrdpic*** which uses the rounding mode specified by RN.

VSR Data Layout for xscvdpuxws



	FPSCR.VE	FPSCR.XE	Inexact? (trunc(src) ≠ src)	Returned Results and Status Setting
src ≤ Nmin−1	0	–	–	T(Nmin), fr(0), fi(0), fx(VXCVI)
	1	–	–	fr(0), fi(0), fx(VXCVI), error()
Nmin−1 < src < Nmin	–	0	yes	T(Nmin), fr(0), fi(1), fx(XX)
		1	yes	T(Nmin), fr(0), fi(1), fx(XX), error()
src = Nmin	–	–	no	T(Nmin), fr(0), fi(0)
Nmin < src < Nmax	–	–	no	T(f2i(trunc(src))), fr(0), fi(0)
		0	yes	T(f2i(trunc(src))), fr(0), fi(1), fx(XX)
		1	yes	T(f2i(trunc(src))), fr(0), fi(1), fx(XX), error()
src = Nmax	–	–	no	T(Nmax), fr(0), fi(0)
Nmax < src < Nmax+1	–	0	yes	T(Nmax), fr(0), fi(1), fx(XX)
		1	yes	T(Nmax), fr(0), fi(1), fx(XX), error()
src ≥ Nmax+1	0	–	–	T(Nmax), fr(0), fi(0), fx(VXCVI)
	1	–	–	fr(0), fi(0), fx(VXCVI), error()
src is a QNaN	0	–	–	T(Nmin), fr(0), fi(0), fx(VXCVI)
	1	–	–	fr(0), fi(0), fx(VXCVI), error()
src is a SNaN	0	–	–	T(Nmin), fr(0), fi(0), fx(VXCVI), fx(VXSNAN)
	1	–	–	fr(0), fi(0), fx(VXCVI), fx(VXSNAN), error()

Explanation:

error()	The system error handler is invoked for the trap-enabled exception if MSR.FE0 and MSR.FE1 bits are set to any mode other than the ignore-exception mode.
f2i(x)	The double-precision floating-point integer value x is converted to 32-bit unsigned integer format.
fi(x)	FPSCR.FI is set to the value x.
fr(x)	FPSCR.FR is set to the value x.
fx(x)	FPSCR.FX is set to 1 if FPSCR.x=0. FPSCR.x is set to 1.
Nmin	The smallest unsigned integer word value, 0 (0x0000_0000).
Nmax	The largest unsigned integer word value, $2^{32} - 1$ (0xFFFF_FFFF).
src	The double-precision floating-point value in doubleword element 0 of VSR[XB].
trunc(x)	The double-precision floating-point value x is truncated to a floating-point integer.
T(x)	The unsigned integer word value x is placed in word elements 0 and 1 of VSR[XT]. The contents of word elements 2 and 3 of VSR[XT] are set to 0.

Table 7.77: Actions for xscvdpuxws

VSX Scalar Convert with round to zero Quad-Precision to Signed Doubleword format X-form

xscvqpsdz VRT,VRB

63	VRT	25	VRB	836	/
0	6	11	16	21	31

```
if MSR.VSX=0 then VSX_Unavailable()

reset_xflags()

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
end
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
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, +2^63-1) then do
      result ← 0x7FFF_FFFF_FFFF_FFFF
      vxcvi_flag ← 1
   end
   else if bfp_COMPARE_LT(rnd, -2^63) then do
      result ← 0x8000_0000_0000_0000
      vxcvi_flag ← 1
   end
   else do
      result ← si64_CONVERT_FROM_BFP(rnd)
      if xx_flag=1 then SetFX(FPSCR.XX)
   end
end

if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
if vxcvi_flag=1  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 ← 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 Quiet NaN or an Infinity, an Invalid Operation exception occurs and VXCVI is set to 1.

If src is a NaN, the result is 0x8000_0000_0000_0000.

Otherwise, if src is a Zero, the result is 0x0000_0000_0000_0000.

Otherwise, if src is $+$Infinity, the result is 0x7FFF_FFFF_FFFF_FFFF.

Otherwise, if src is $-$Infinity, the result is 0x8000_0000_0000_0000.

Otherwise, do the following.

Let rnd be the value src truncated to a floating-point integer.

If rnd is greater than $+2^{63} - 1$, an Invalid Operation exception occurs, VXCVI is set to 1, and the result is 0x7FFF_FFFF_FFFF_FFFF.

Otherwise, if rnd is less than -2^{63}, an Invalid Operation exception occurs, VXCVI is set to 1, and the result is 0x8000_0000_0000_0000.

Otherwise, the result is the value rnd, and an Inexact exception occurs if rnd is inexact (i.e., rnd is not equal to src).

The result is placed into doubleword element 0 of VSR[VRT+32] in signed integer format.

The contents of doubleword element 1 of VSR[VRT+32] are set to 0.

FPRF is set to undefined. FR is set to 0. FI is set to indicate if the rounded result is inexact.

If an Invalid Operation exception occurs, FR and FI are set to 0.

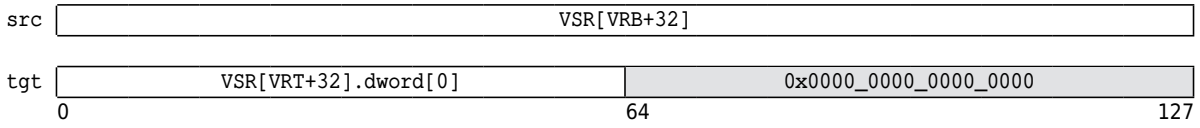
If a trap-enabled Invalid Operation exception occurs, VSR[VRT+32] and FPRF are not modified.

See Table 7.78, “Actions for xscvqpsdz,” on page 825.

Special Registers Altered:

Register	*Field(s)*
FPSCR	FPRF (undefined)
FPSCR	FR FI FX VXSNAN VXCVI XX

VSR Data Layout for xscvqpsdz



	FPSCR.VE	FPSCR.XE	Inexact? (trunc(src) ≠ src)	Returned Results and Status Setting
src ≤ Nmin−1	0	–	–	T(Nmin), fr(0), fi(0), fprf(0bUUUUU), fx(VXCVI)
	1	–	–	fr(0), fi(0), fx(VXCVI), error()
Nmin−1 < src < Nmin	–	0	yes	T(Nmin), fr(0), fi(1), fprf(0bUUUUU), fx(XX)
		1	yes	T(Nmin), fr(0), fi(1), fprf(0bUUUUU), fx(XX), error()
src = Nmin	–	–	no	T(Nmin), fr(0), fi(0), fprf(0bUUUUU)
Nmin < src < Nmax	–	–	no	T(f2i(trunc(src))), fr(0), fi(0), fprf(0bUUUUU)
		0	yes	T(f2i(trunc(src))), fr(0), fi(1), fprf(0bUUUUU), fx(XX)
		1	yes	T(f2i(trunc(src))), fr(0), fi(1), fprf(0bUUUUU), fx(XX), error()
src = Nmax	–	–	no	T(Nmax), fr(0), fi(0), fprf(0bUUUUU)
Nmax < src < Nmax+1	–	0	yes	T(Nmax), fr(0), fi(1), fprf(0bUUUUU), fx(XX)
		1	yes	T(Nmax), fr(0), fi(1), fprf(0bUUUUU), fx(XX), error()
src ≥ Nmax+1	0	–	–	T(Nmax), fr(0), fi(0), fprf(0bUUUUU), fx(VXCVI)
	1	–	–	fr(0), fi(0), fx(VXCVI), error()
src is a QNaN	0	–	–	T(Nmin), fr(0), fi(0), fprf(0bUUUUU), fx(VXCVI)
	1	–	–	fr(0), fi(0), fx(VXCVI), error()
src is a SNaN	0	–	–	T(Nmin), fr(0), fi(0), fprf(0bUUUUU), fx(VXCVI), fx(VXSNAN)
	1	–	–	fr(0), fi(0), fx(VXCVI), fx(VXSNAN), error()

Explanation:

T(x)	Places the value x into the target VSR. VSR[VRT+32].dword[0] ← x VSR[VRT+32].dword[1] ← 0x0000_0000_0000_0000
Nmin	The smallest 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].
f2i(x)	The quad-precision floating-point integer value x is converted to 64-bit signed integer format.
fi(x)	FPSCR.FI is set to the value x.
fprf(x)	FPSCR.FPRF is set to the value x.
fr(x)	FPSCR.FR is set to the value x.
fx(x)	FPSCR.FX is set to 1 if FPSCR.x=0. FPSCR.x is set to 1.
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 quad-precision floating-point value x truncated to a floating-point integer.

Table 7.78: Actions for xscvqpsdz

VSX Scalar Convert with round to zero Quad-Precision to Signed Quadword X-form

xscvqpsqz VRT,VRB

63	VRT	8	VRB	836	/
0	6	11	16	21	31

```
if MSR.VSX=0 then VSX_Unavailable()

reset_xflags()

src ← bfp_CONVERT_FROM_BFP128(VSR[VRB+32])

if src.class.QNaN=1 | src.class.SNaN=1 then do
   result ← 0x8000_0000_0000_0000_0000_0000_0000_0000
   vxsnan_flag ← src.class.SNaN
   vxcvi_flag ← 1
end
else if src.class.Infinity=1 then do
   vxcvi_flag ← 1
   if src.sign = 0 then
     result ← 0x7FFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF
   else
     result ← 0x8000_0000_0000_0000_0000_0000_0000_0000
end
else if src.class.Zero=1 then
   result ← 0x0000_0000_0000_0000_0000_0000_0000_0000
else do
   rnd ← bfp_ROUND_TO_INTEGER(0b001,src)
   if bfp_COMPARE_GT(rnd, +2^127-1) then do
     result ← 0x7FFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF
     vxcvi_flag ← 1
   end
   else if bfp_COMPARE_LT(rnd, -2^127) then do
     result ← 0x8000_0000_0000_0000_0000_0000_0000_0000
     vxcvi_flag ← 1
   end
   else do
     result ← si128_CONVERT_FROM_BFP(rnd)
     if xx_flag=1 then SetFX(FPSCR.XX)
   end
end

if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
if vxcvi_flag=1  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] ← result
   FPSCR.FPRF ← 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 Quiet NaN or an Infinity, an Invalid Operation exception occurs and VXCVI is set to 1.

If src is a NaN, the result is -2^{127}.

Otherwise, if src is a Zero, the result is 0.

Otherwise, if src is +Infinity, the result is $2^{127}-1$.

Otherwise, if src is −Infinity, the result is -2^{127}.

Otherwise, do the following.

Let rnd be the value src truncated to a floating-point integer.

If rnd is greater than $+2^{127}-1$, an Invalid Operation exception occurs, VXCVI is set to 1, and the result is 2^{127}.

Otherwise, if rnd is less than -2^{127}, an Invalid Operation exception occurs, VXCVI is set to 1, and the result is -2^{127}.

Otherwise, the result is the value 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 7.79, “Actions for xscvqpsqz,” on page 827.

Special Registers Altered:

Register	*Field(s)*
FPSCR	FPRF (undefined)
FPSCR	FR (set to 0)
FPSCR	FI FX VXSNAN VXCVI XX

VSR Data Layout for xscvqpsqz

	FPSCR.VE	FPSCR.XE	Inexact? (trunc(src) ≠ src)	Returned Results and Status Setting
src ≤ Nmin−1	0	-	-	T(Nmin), fr(0), fi(0), fprf(0bUUUUU), fx(VXCVI)
	1	-	-	fr(0), fi(0), fx(VXCVI), error()
Nmin−1 < src < Nmin	-	0	yes	T(Nmin), fr(0), fi(1), fprf(0bUUUUU), fx(XX)
		1	yes	T(Nmin), fr(0), fi(1), fprf(0bUUUUU), fx(XX), error()
src = Nmin	-	-	no	T(Nmin), fr(0), fi(0), fprf(0bUUUUU)
Nmin < src < Nmax	-	-	no	T(f2i(trunc(src))), fr(0), fi(0), fprf(0bUUUUU)
		0	yes	T(f2i(trunc(src))), fr(0), fi(1), fprf(0bUUUUU), fx(XX)
		1	yes	T(f2i(trunc(src))), fr(0), fi(1), fprf(0bUUUUU), fx(XX), error()
src = Nmax	-	-	no	T(Nmax), fr(0), fi(0), fprf(0bUUUUU)
Nmax < src < Nmax+1	-	0	yes	T(Nmax), fr(0), fi(1), fprf(0bUUUUU), fx(XX)
		1	yes	T(Nmax), fr(0), fi(1), fprf(0bUUUUU), fx(XX), error()
src ≥ Nmax+1	0	-	-	T(Nmax), fr(0), fi(0), fprf(0bUUUUU), fx(VXCVI)
	1	-	-	fr(0), fi(0), fx(VXCVI), error()
src is a QNaN	0	-	-	T(Nmin), fr(0), fi(0), fprf(0bUUUUU), fx(VXCVI)
	1	-	-	fr(0), fi(0), fx(VXCVI), error()
src is a SNaN	0	-	-	T(Nmin), fr(0), fi(0), fprf(0bUUUUU), fx(VXCVI), fx(VXSNAN)
	1	-	-	fr(0), fi(0), fx(VXCVI), fx(VXSNAN), error()

Explanation:

T(x)	Places the value x into the target VSR.
Nmin	The smallest signed integer doubleword value, -2^{127} (0x8000_0000_0000_0000_0000_0000_0000_0000).
Nmax	The largest signed integer doubleword value, $2^{127}-1$ (0x7FFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF).
src	The quad-precision floating-point value in VSR[VRB+32].
f2i(x)	The quad-precision floating-point integer value x is converted to 128-bit signed integer format.
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 7.79: Actions for xscvqpsqz

VSX Scalar Convert with round to zero Quad-Precision to Signed Word format X-form

xscvqpswz VRT,VRB

63	VRT	9	VRB	836	/
0	6	11	16	21	31

```
if MSR.VSX=0 then VSX_Unavailable()

reset_xflags()

src ← bfp_CONVERT_FROM_BFP128(VSR[VRB+32])

if src.class.QNaN=1 | src.class.SNaN=1 then do
   result ← 0xFFFF_FFFF_8000_0000
   vxsnan_flag ← src.class.SNaN
   vxcvi_flag ← 1
end
else if src.class.Infinity=1 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=1 then
   result ← 0x0000_0000_0000_0000
else do
   rnd ← bfp_ROUND_TO_INTEGER(0b001,src)
   if bfp_COMPARE_GT(rnd, +2^31-1) then do
      result ← 0x0000_0000_7FFF_FFFF
      vxcvi_flag ← 1
   end
   else if bfp_COMPARE_LT(rnd, -2^31) then do
      result ← 0xFFFF_FFFF_8000_0000
      vxcvi_flag ← 1
   end
   else do
      result ← si64_CONVERT_FROM_BFP(rnd)
      if xx_flag=1 then SetFX(FPSCR.XX)
   end
end

if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
if vxcvi_flag=1  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 ← 0bUUUUU
end
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 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.

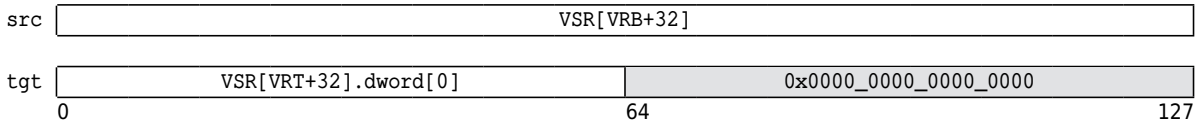
If a trap-enabled Invalid Operation exception occurs, VSR[VRT+32] and FPRF are not modified.

See Table 7.80, “Actions for xscvqpswz,” on page 829.

Special Registers Altered:

Register	*Field(s)*
FPSCR	FPRF (undefined)
FPSCR	FR (set to 0)
FPSCR	FI FX VXSNAN VXCVI XX

VSR Data Layout for xscvqpswz



	FPSCR.VE	FPSCR.XE	Inexact? (trunc(src) ≠ src)	Returned Results and Status Setting
src ≤ Nmin−1	0	–	–	T(Nmin), fr(0), fi(0), fprf(0bUUUUU), fx(VXCVI)
	1	–	–	fr(0), fi(0), fx(VXCVI), error()
Nmin−1 < src < Nmin	–	0	yes	T(Nmin), fr(0), fi(1), fprf(0bUUUUU), fx(XX)
		1	yes	T(Nmin), fr(0), fi(1), fprf(0bUUUUU), fx(XX), error()
src = Nmin	–	–	no	T(Nmin), fr(0), fi(0), fprf(0bUUUUU)
Nmin < src < Nmax	–	–	no	T(f2i(trunc(src))), fr(0), fi(0), fprf(0bUUUUU)
		0	yes	T(f2i(trunc(src))), fr(0), fi(1), fprf(0bUUUUU), fx(XX)
		1	yes	T(f2i(trunc(src))), fr(0), fi(1), fprf(0bUUUUU), fx(XX), error()
src = Nmax	–	–	no	T(Nmax), fr(0), fi(0), fprf(0bUUUUU)
Nmax < src < Nmax+1	–	0	yes	T(Nmax), fr(0), fi(1), fprf(0bUUUUU), fx(XX)
		1	yes	T(Nmax), fr(0), fi(1), fprf(0bUUUUU), fx(XX), error()
src ≥ Nmax+1	0	–	–	T(Nmax), fr(0), fi(0), fprf(0bUUUUU), fx(VXCVI)
	1	–	–	fr(0), fi(0), fx(VXCVI), error()
src is a QNaN	0	–	–	T(Nmin), fr(0), fi(0), fprf(0bUUUUU), fx(VXCVI)
	1	–	–	fr(0), fi(0), fx(VXCVI), error()
src is a SNaN	0	–	–	T(Nmin), fr(0), fi(0), fprf(0bUUUUU), fx(VXCVI), fx(VXSNAN)
	1	–	–	fr(0), fi(0), fx(VXCVI), fx(VXSNAN), error()

Explanation:

T(x)	Places the value x into the target VSR. VSR[VRT+32].dword[0] ← x VSR[VRT+32].dword[1] ← 0x0000_0000_0000_0000
Nmin	The smallest signed integer word value, -2^{31} (0xFFFF_FFFF_8000_0000).
Nmax	The largest signed integer word value, $2^{31}-1$ (0x0000_0000_7FFF_FFFF).
src	The quad-precision floating-point value in VSR[VRB+32].
f2i(x)	The quad-precision floating-point integer value x is converted to 32-bit signed integer format.
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 7.80: Actions for xscvqpswz

VSX Scalar Convert with round to zero Quad-Precision to Unsigned Doubleword format X-form

xscvqpudz VRT,VRB

63	VRT	17	VRB	836	/
0	6	11	16	21	31

```
if MSR.VSX=0 then VSX_Unavailable()

reset_xflags()

src ← bfp_CONVERT_FROM_BFP128(VSR[VRB+32])

if src.class.QNaN=1 | src.class.SNaN=1 then do
   result ← 0x0000_0000_0000_0000
   vxsnan_flag ← src.class.SNaN
   vxcvi_flag ← 1
end
else if src.class.Infinity=1 then do
   vxcvi_flag ← 1
   if src.sign=0 then
      result ← 0xFFFF_FFFF_FFFF_FFFF
   else
      result ← 0x0000_0000_0000_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, +2^64-1) then do
      result ← 0xFFFF_FFFF_FFFF_FFFF
      vxcvi_flag ← 1
   end
   else if bfp_COMPARE_LT(rnd, 0) then do
      result ← 0x0000_0000_0000_0000
      vxcvi_flag ← 1
   end
   else do
      result ← ui64_CONVERT_FROM_BFP(rnd)
      if xx_flag=1 then SetFX(FPSCR.XX)
   end
end

if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
if vxcvi_flag=1  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 ← 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 Quiet NaN or an Infinity, an Invalid Operation exception occurs and VXCVI is set to 1.

If src is a NaN, the result is 0x0000_0000_0000_0000.

Otherwise, if src is a Zero, the result is 0x0000_0000_0000_0000.

Otherwise, if src is a positive Infinity, the result is 0xFFFF_FFFF_FFFF_FFFF.

Otherwise, if src is a negative Infinity, the result is 0x0000_0000_0000_0000.

Otherwise, do the following.

Let rnd be the value src truncated to a floating-point integer.

If rnd is greater than $+2^{64}-1$, an Invalid Operation exception occurs, VXCVI is set to 1, and the result is 0xFFFF_FFFF_FFFF_FFFF.

Otherwise, if rnd is less than 0, an Invalid Operation exception occurs, VXCVI is set to 1, and the result is 0x0000_0000_0000_0000.

Otherwise, the result is the value rnd, and an Inexact exception occurs if rnd is inexact (i.e., rnd is not equal to src).

The result is placed into doubleword element 0 of VSR[VRT+32] in unsigned integer format.

The contents of doubleword element 1 of VSR[VRT+32] are set to 0.

FPRF is set to undefined. FR is set to 0. FI is set to indicate if the rounded result is inexact.

If an Invalid Operation exception occurs, FR and FI are set to 0.

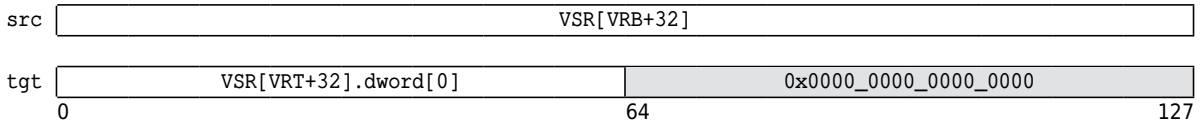
If a trap-enabled Invalid Operation exception occurs, VSR[VRT+32] and FPRF are not modified.

See Table 7.81, “Actions for xscvqpudz,” on page 831.

Special Registers Altered:

Register	*Field(s)*
FPSCR	FPRF (undefined)
FPSCR	FR (set to 0)
FPSCR	FI FX VXSNAN VXCVI XX

VSR Data Layout for xscvqpudz



	FPSCR.VE	FPSCR.XE	Inexact? (trunc(src) ≠ src)	Returned Results and Status Setting
src ≤ Nmin−1	0	–	–	T(Nmin), fr(0), fi(0), fprf(0bUUUUU), fx(VXCVI)
	1	–	–	fr(0), fi(0), fx(VXCVI), error()
Nmin−1 < src < Nmin	–	0	yes	T(Nmin), fr(0), fi(1), fprf(0bUUUUU), fx(XX)
		1	yes	T(Nmin), fr(0), fi(1), fprf(0bUUUUU), fx(XX), error()
src = Nmin	–	–	no	T(Nmin), fr(0), fi(0), fprf(0bUUUUU)
Nmin < src < Nmax	–	–	no	T(f2i(trunc(src))), fr(0), fi(0), fprf(0bUUUUU)
		0	yes	T(f2i(trunc(src))), fr(0), fi(1), fprf(0bUUUUU), fx(XX)
		1	yes	T(f2i(trunc(src))), fr(0), fi(1), fprf(0bUUUUU), fx(XX), error()
src = Nmax	–	–	no	T(Nmax), fr(0), fi(0), fprf(0bUUUUU)
Nmax < src < Nmax+1	–	0	yes	T(Nmax), fr(0), fi(1), fprf(0bUUUUU), fx(XX)
		1	yes	T(Nmax), fr(0), fi(1), fprf(0bUUUUU), fx(XX), error()
src ≥ Nmax+1	0	–	–	T(Nmax), fr(0), fi(0), fprf(0bUUUUU), fx(VXCVI)
	1	–	–	fr(0), fi(0), fx(VXCVI), error()
src is a QNaN	0	–	–	T(Nmin), fr(0), fi(0), fprf(0bUUUUU), fx(VXCVI)
	1	–	–	fr(0), fi(0), fx(VXCVI), error()
src is a SNaN	0	–	–	T(Nmin), fr(0), fi(0), fprf(0bUUUUU), fx(VXCVI), fx(VXSNAN)
	1	–	–	fr(0), fi(0), fx(VXCVI), fx(VXSNAN), error()

Explanation:

T(x)	Places the value x into the target VSR. VSR[VRT+32].dword[0] ← x VSR[VRT+32].dword[1] ← 0x0000_0000_0000_0000
Nmin	The smallest unsigned integer doubleword value, 0 (0x0000_0000_0000_0000).
Nmax	The largest unsigned integer doubleword value, $2^{64} - 1$ (0xFFFF_FFFF_FFFF_FFFF).
src	The quad-precision floating-point value in VSR[VRB+32].
f2i(x)	The quad-precision floating-point integer value x is converted to 64-bit unsigned integer format.
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 quad-precision floating-point value x truncated to a floating-point integer.

Table 7.81: Actions for xscvqpudz

VSX Scalar Convert with round to zero Quad-Precision to Unsigned Quadword X-form

xscvqpuqz VRT,VRB

63	VRT	0	VRB	836	/
0	6	11	16	21	31

```
if MSR.VSX=0 then VSypX_Unavailable()

reset_xflags()

src ← bfp_CONVERT_FROM_BFP128(VSR[VRB+32])

if src.class.QNaN=1 | src.class.SNaN=1 then do
   result ← 0x0000_0000_0000_0000_0000_0000_0000_0000
   vxsnan_flag ← src.class.SNaN
   vxcvi_flag ← 1
end
else if src.class.Infinity=1 then do
   vxcvi_flag ← 1
   if src.sign=0 then
     result ← 0xFFFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF
   else
     result ← 0x0000_0000_0000_0000_0000_0000_0000_0000
end
else if src.class.Zero=1 then
   result ← 0x0000_0000_0000_0000_0000_0000_0000_0000
else do
   rnd ← bfp_ROUND_TO_INTEGER(0b001,src)
   if bfp_COMPARE_GT(rnd, +2^128-1) then do
     result ← 0xFFFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF
     vxcvi_flag ← 1
   end
   else if bfp_COMPARE_LT(rnd, 0) then do
     result ← 0x0000_0000_0000_0000_0000_0000_0000_0000
     vxcvi_flag ← 1
   end
   else do
     result ← si128_CONVERT_FROM_BFP(rnd)
if xx_flag=1 then SetFX(FPSCR.XX)
   end
end

if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
if vxcvi_flag=1  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] ← result
   FPSCR.FPRF ← 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 Quiet NaN or an Infinity, an Invalid Operation exception occurs and VXCVI is set to 1.

If src is a NaN, the result is 0.

Otherwise, if src is a Zero, the result is 0.

Otherwise, if src is +Infinity, the result is 2^{128}-1.

Otherwise, if src is −Infinity, the result is 0.

Otherwise, do the following.

Let rnd be the value src truncated to a floating-point integer.

If rnd is greater than $+2^{128}-1$, an Invalid Operation exception occurs, VXCVI is set to 1, and the result is $+2^{128}-1$.

Otherwise, if rnd is less than 0, an Invalid Operation exception occurs, VXCVI is set to 1, and the result is 0.

Otherwise, the result is the value rnd, and an Inexact exception occurs if rnd is inexact (i.e., rnd is not equal to src).

The result is placed into VSR[VRT+32] in unsigned integer format.

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 7.82, “Actions for xscvqpuqz,” on page 833.

Special Registers Altered:

Register	*Field(s)*
FPSCR	FPRF (undefined)
FPSCR	FR (set to 0)
FPSCR	FI FX VXSNAN VXCVI XX

VSR Data Layout for xscvqpuqz

	FPSCR.VE	FPSCR.XE	Inexact? (trunc(src) ≠ src)	Returned Results and Status Setting
src ≤ Nmin−1	0	-	-	T(Nmin), fr(0), fi(0), fprf(0bUUUUU), fx(VXCVI)
	1	-	-	fr(0), fi(0), fx(VXCVI), error()
Nmin−1 < src < Nmin	-	0	yes	T(Nmin), fr(0), fi(1), fprf(0bUUUUU), fx(XX)
		1	yes	T(Nmin), fr(0), fi(1), fprf(0bUUUUU), fx(XX), error()
src = Nmin	-	-	no	T(Nmin), fr(0), fi(0), fprf(0bUUUUU)
Nmin < src < Nmax	-	-	no	T(f2i(trunc(src))), fr(0), fi(0), fprf(0bUUUUU)
		0	yes	T(f2i(trunc(src))), fr(0), fi(1), fprf(0bUUUUU), fx(XX)
		1	yes	T(f2i(trunc(src))), fr(0), fi(1), fprf(0bUUUUU), fx(XX), error()
src = Nmax	-	-	no	T(Nmax), fr(0), fi(0), fprf(0bUUUUU)
Nmax < src < Nmax+1	-	0	yes	T(Nmax), fr(0), fi(1), fprf(0bUUUUU), fx(XX)
		1	yes	T(Nmax), fr(0), fi(1), fprf(0bUUUUU), fx(XX), error()
src ≥ Nmax+1	0	-	-	T(Nmax), fr(0), fi(0), fprf(0bUUUUU), fx(VXCVI)
	1	-	-	fr(0), fi(0), fx(VXCVI), error()
src is a QNaN	0	-	-	T(Nmin), fr(0), fi(0), fprf(0bUUUUU), fx(VXCVI)
	1	-	-	fr(0), fi(0), fx(VXCVI), error()
src is a SNaN	0	-	-	T(Nmin), fr(0), fi(0), fprf(0bUUUUU), fx(VXCVI), fx(VXSNAN)
	1	-	-	fr(0), fi(0), fx(VXCVI), fx(VXSNAN), error()

Explanation:

T(x)	Places the value x into the target VSR.
Nmin	The smallest unsigned integer doubleword value, 0 (0x0000_0000_0000_0000_0000_0000_0000_0000).
Nmax	The largest unsigned integer doubleword value, $2^{128} - 1$ (0xFFFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF).
src	The quad-precision floating-point value in VSR[VRB+32].
f2i(x)	The quad-precision floating-point integer value x is converted to 128-bit unsigned integer format.
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 7.82: Actions for xscvqpuqz

VSX Scalar Convert with round to zero Quad-Precision to Unsigned Word format X-form

xscvqpuwz VRT,VRB

63	VRT	1	VRB	836	/
0	6	11	16	21	31

```
if MSR.VSX=0 then VSX_Unavailable()

reset_xflags()

src ← bfp_CONVERT_FROM_BFP128(VSR[VRB+32])

if src.class.QNaN=1 | src.class.SNaN=1 then do
   result ← 0x0000_0000
   vxsnan_flag ← src.class.SNaN
   vxcvi_flag ← 1
end
else if src.class.Infinity=1 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=1 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 ← ui64_CONVERT_FROM_BFP(rnd)
      if xx_flag=1 then SetFX(FPSCR.XX)
   end
end

if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
if vxcvi_flag=1  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 ← 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 Quiet NaN or an Infinity, an Invalid Operation exception occurs and VXCVI is set to 1.

If src is a NaN, the result is 0x0000_0000_0000_0000.

Otherwise, if src is a Zero, the result is 0x0000_0000_0000_0000.

Otherwise, if src is a positive Infinity, the result is 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.

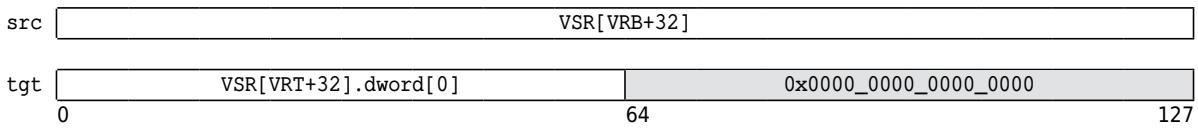
If a trap-enabled Invalid Operation exception occurs, VSR[VRT+32] and FPRF are not modified.

See Table 7.83, “Actions for xscvqpuwz,” on page 835.

Special Registers Altered:

Register	*Field(s)*
FPSCR	FPRF (undefined)
FPSCR	FR (set to 0)
FPSCR	FI FX VXSNAN VXCVI XX

VSR Data Layout for xscvqpuwz



	FPSCR.VE	FPSCR.XE	Inexact? (trunc(src) ≠ src)	Returned Results and Status Setting
src ≤ Nmin−1	0	-	-	T(Nmin), fr(0), fi(0), fprf(0bUUUUU), fx(VXCVI)
	1	-	-	fr(0), fi(0), fx(VXCVI), error()
Nmin−1 < src < Nmin	-	0	yes	T(Nmin), fr(0), fi(1), fprf(0bUUUUU), fx(XX)
		1	yes	T(Nmin), fr(0), fi(1), fprf(0bUUUUU), fx(XX), error()
src = Nmin	-	-	no	T(Nmin), fr(0), fi(0), fprf(0bUUUUU)
Nmin < src < Nmax	-	-	no	T(f2i(trunc(src))), fr(0), fi(0), fprf(0bUUUUU)
		0	yes	T(f2i(trunc(src))), fr(0), fi(1), fprf(0bUUUUU), fx(XX)
		1	yes	T(f2i(trunc(src))), fr(0), fi(1), fprf(0bUUUUU), fx(XX), error()
src = Nmax	-	-	no	T(Nmax), fr(0), fi(0), fprf(0bUUUUU)
Nmax < src < Nmax+1	-	0	yes	T(Nmax), fr(0), fi(1), fprf(0bUUUUU), fx(XX)
		1	yes	T(Nmax), fr(0), fi(1), fprf(0bUUUUU), fx(XX), error()
src ≥ Nmax+1	0	-	-	T(Nmax), fr(0), fi(0), fprf(0bUUUUU), fx(VXCVI)
	1	-	-	fr(0), fi(0), fx(VXCVI), error()
src is a QNaN	0	-	-	T(Nmin), fr(0), fi(0), fprf(0bUUUUU), fx(VXCVI)
	1	-	-	fr(0), fi(0), fx(VXCVI), error()
src is a SNaN	0	-	-	T(Nmin), fr(0), fi(0), fprf(0bUUUUU), fx(VXCVI), fx(VXSNAN)
	1	-	-	fr(0), fi(0), fx(VXCVI), fx(VXSNAN), error()

Explanation:

T(x)	Places the value x into the target VSR. VSR[VRT+32].dword[0] ← x VSR[VRT+32].dword[1] ← 0x0000_0000_0000_0000
Nmin	The smallest unsigned integer word value, 0 (0x0000_0000_0000_0000).
Nmax	The largest unsigned integer word value, $2^{32} - 1$ (0x0000_0000_FFFF_FFFF).
src	The quad-precision floating-point value in VSR[VRB+32].
f2i(x)	The quad-precision floating-point integer value x is converted to 32-bit unsigned integer format.
fx(x)	FPSCR.FX is set to 1 if FPSCR.x=0. FPSCR.x is set to 1.
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 7.83: Actions for xscvqpuwz

7.6.2.9.2 VSX Vector BFP Convert To Integer Instructions

VSX Vector Convert with round to zero Double-Precision to Signed Doubleword format XX2-form

xvcvdpsxds XT,XB

60	T	///	B	472	BX	TX
0	6	11	16	21	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

ex_flag ← 0b0

do i = 0 to 1
   reset_xflags()

   src ← bfp_CONVERT_FROM_BFP64(VSR[32×BX+B].dword[i])
   rnd ← bfp_ROUND_TO_INTEGER(0b001, src)

   vresult.dword[i] ← si64_CONVERT_FROM_BFP(rnd)

   if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
   if vxcvi_flag=1 then SetFX(FPSCR.VXCVI)
   if xx_flag=1 then SetFX(FPSCR.XX)

   ex_flag ← ex_flag | (FPSCR.VE & vxsnan_flag)
                     | (FPSCR.VE & vxcvi_flag)
                     | (FPSCR.XE & xx_flag)
end

if ex_flag=0 then VSR[32×TX+T] ← result
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

For each integer value i from 0 to 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 $2^{63}-1$, the result is 0x7FFF_FFFF_FFFF_FFFF and VXCVI is set to 1.

Otherwise, if the rounded value is less than -2^{63}, the result is 0x8000_0000_0000_0000 and VXCVI is set to 1.

Otherwise, the result is the rounded value converted to 64-bit signed-integer format, and if the result is inexact (i.e., not equal to src), XX is set to 1.

The result is placed into doubleword element i of VSR[XT].

See Table 7.84.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

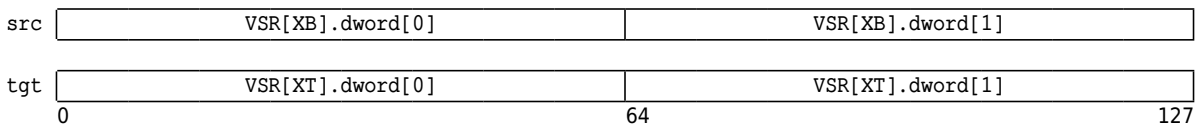
Special Registers Altered:

Register	*Field(s)*
FPSCR	FX XX VXSNAN VXCVI

Programming Note

xvcvdpsxds 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 the RN.

VSR Data Layout for xvcvdpsxds



	FPSCR.VE	FPSCR.XE	Inexact? (trunc(src) ≠ src)	Returned Results and Status Setting
src ≤ Nmin−1	0	–	–	T(Nmin), fx(VXCVI)
	1	–	–	fx(VXCVI), error()
Nmin−1 < src < Nmin	–	0	yes	T(Nmin), fx(XX)
		1	yes	fx(XX), error()
src = Nmin	–	–	no	T(Nmin)
Nmin < src < Nmax	–	–	no	T(f2i(trunc(src)))
		0	yes	T(f2i(trunc(src))), fx(XX)
		1	yes	fx(XX), error()
src = Nmax	–	–	no	T(Nmax) Note: This case cannot occur as Nmax is not representable in DP format but is included here for completeness.
Nmax < src < Nmax+1	–	0	yes	T(Nmax), fx(XX)
		1	yes	fx(XX), error()
src ≥ Nmax+1	0	–	–	T(Nmax), fx(VXCVI)
	1	–	–	fx(VXCVI), error()
src is a QNaN	0	–	–	T(Nmin), fx(VXCVI)
	1	–	–	fx(VXCVI), error()
src is a SNaN	0	–	–	T(Nmin), fx(VXCVI), fx(VXSNAN)
	1	–	–	fx(VXCVI), error(), fx(VXSNAN)

Explanation:

error()	The system error handler is invoked for the trap-enabled exception if MSR.FE0 and MSR.FE1 bits are set to any mode other than the ignore-exception mode. Update of VSR[XT] is suppressed.
f2i(x)	The double-precision floating-point integer value x is converted to 64-bit signed integer format.
fx(x)	FPSCR.FX is set to 1 if FPSCR.x=0. FPSCR.x is set to 1.
Nmin	The smallest signed integer doubleword value, -2^{63} (0x8000_0000_0000_0000).
Nmax	The largest signed integer doubleword value, $2^{63}-1$ (0x7FFF_FFFF_FFFF_FFFF).
src	The double-precision floating-point value in doubleword element 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}).
trunc(x)	The double-precision floating-point value x is truncated to a floating-point integer.

Table 7.84: Actions for xvcvdpsxds

VSX Vector Convert with round to zero Double-Precision to Signed Word format XX2-form

xvcvdpsxws XT,XB

60	T	///	B	216	BX	TX
0	6	11	16	21	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

ex_flag ← 0b0

do i = 0 to 1
   reset_xflags()

   src ← bfp_CONVERT_FROM_BFP64(VSR[32×BX+B].dword[i])
   rnd ← bfp_ROUND_TO_INTEGER(0b001, src)

   vresult.dword[i].word[0] ← si32_CONVERT_FROM_BFP(rnd)
   vresult.dword[i].word[1] ← si32_CONVERT_FROM_BFP(rnd)

   if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
   if vxcvi_flag=1 then SetFX(FPSCR.VXCVI)
   if xx_flag=1 then SetFX(FPSCR.XX)

   ex_flag ← ex_flag | (FPSCR.VE & vxsnan_flag)
                     | (FPSCR.VE & vxcvi_flag)
                     | (FPSCR.XE & xx_flag)
end

if ex_flag=0 then VSR[32×TX+T] ← vresult
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

For each integer value i from 0 to 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 $2^{31}-1$, 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 bits 0:31 of doubleword element i of VSR[XT].

The result is also placed into bits 32:63 of doubleword element i of VSR[XT].

See Table 7.85.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered:

Register	*Field(s)*
FPSCR	FX XX VXSNAN VXCVI

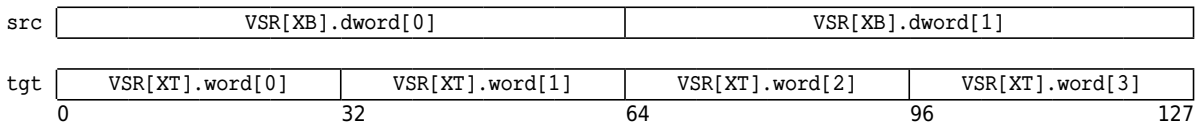
Programming Note

xvcvdpsxws 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.

Programming Note

Previous versions of the architecture allowed the contents of words 1 and 3 of the result register to be undefined. However, all processors that support this instruction write the result into words 0 and 1 and words 2 and 3 of the result register, as is required by this version of the architecture.

VSR Data Layout for xvcvdpsxws



	FPSCR.VE	FPSCR.XE	Inexact? (trunc(src) ≠ src)	Returned Results and Status Setting
src ≤ Nmin-1	0	-	-	T(Nmin), fx(VXCVI)
	1	-	-	fx(VXCVI), error()
Nmin-1 < src < Nmin	-	0	yes	T(Nmin), fx(XX)
		1	yes	fx(XX), error()
src = Nmin	-	-	no	T(Nmin)
Nmin < src < Nmax	-	-	no	T(f2i(trunc(src)))
		0	yes	T(f2i(trunc(src))), fx(XX)
		1	yes	fx(XX), error()
src = Nmax	-	-	no	T(Nmax)
Nmax < src < Nmax+1	-	0	yes	T(Nmax), fx(XX)
		1	yes	fx(XX), error()
src ≥ Nmax+1	0	-	-	T(Nmax), fx(VXCVI)
	1	-	-	fx(VXCVI), error()
src is a QNaN	0	-	-	T(Nmin), fx(VXCVI)
	1	-	-	fx(VXCVI), error()
src is a SNaN	0	-	-	T(Nmin), fx(VXCVI), fx(VXSNAN)
	1	-	-	fx(VXCVI), fx(VXSNAN), error()

Explanation:

error()	The system error handler is invoked for the trap-enabled exception if MSR.FE0 and MSR.FE1 bits are set to any mode other than the ignore-exception mode. Update of VSR[XT] is suppressed.
f2i(x)	The double-precision floating-point integer value x is converted to 32-bit signed integer format.
fx(x)	FPSCR.FX is set to 1 if FPSCR.x=0. FPSCR.x is set to 1.
Nmin	The smallest signed integer word value, -2^{31} (0x8000_0000).
Nmax	The largest signed integer word value, $2^{31}-1$ (0x7FFF_FFFF).
src	The double-precision floating-point value in doubleword element i of VSR[XB] (where i∈{0,1}).
T(x)	The signed integer word value x is placed in word elements 2×i and 2×i+1 of VSR[XT] (where i∈{0,1}).
trunc(x)	The double-precision floating-point value x is truncated to a floating-point integer.

Table 7.85: Actions for xvcvdpsxws

VSX Vector Convert with round to zero Double-Precision to Unsigned Doubleword format XX2-form

xvcvdpuxds XT,XB

60	T	///	B	456	BX	TX
0	6	11	16	21	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

ex_flag ← 0b0

do i = 0 to 1
   reset_xflags()

   src ← bfp_CONVERT_FROM_BFP64(VSR[32×BX+B].dword[i])
   rnd ← bfp_ROUND_TO_INTEGER(0b001, src)

   vresult.dword[i] ← ui64_CONVERT_FROM_BFP(rnd)

   if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
   if vxcvi_flag=1 then SetFX(FPSCR.VXCVI)
   if xx_flag=1 then SetFX(FPSCR.XX)

   ex_flag ← ex_flag | (FPSCR.VE & vxsnan_flag)
                     | (FPSCR.VE & vxcvi_flag)
                     | (FPSCR.XE & xx_flag)
end

if ex_flag=0 then VSR[32×TX+T] ← vresult
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

For each integer value i from 0 to 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 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 7.86.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

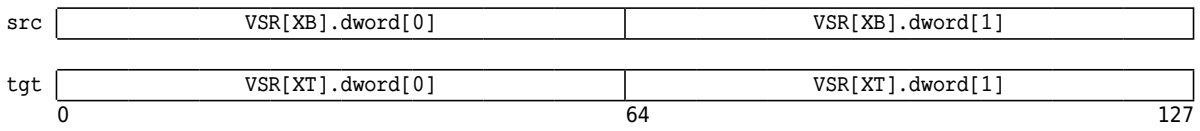
Special Registers Altered:

Register	*Field(s)*
FPSCR	FX XX VXSNAN VXCVI

Programming Note

xvcvdpuxds rounds using Round towards Zero rounding mode. For other rounding modes, software must use a *Round to Double-Precision Integer* instruction that corresponds to the desired rounding mode, including ***xvrdpic*** which uses the rounding mode specified by the RN.

VSR Data Layout for xvcvdpuxds



	FPSCR.VE	FPSCR.XE	Inexact? (trunc(src) ≠ src)	Returned Results and Status Setting
src ≤ Nmin−1	0	–	–	T(Nmin), fx(VXCVI)
	1	–	–	fx(VXCVI), error()
Nmin−1 < src < Nmin	–	0	yes	T(Nmin), fx(XX)
		1	yes	fx(XX), error()
src = Nmin	–	–	no	T(Nmin)
Nmin < src < Nmax	–	–	no	T(f2i(trunc(src)))
		0	yes	T(f2i(trunc(src))), fx(XX)
		1	yes	fx(XX), error()
src = Nmax	–	–	no	T(Nmax) Note: This case cannot occur as Nmax is not representable in DP format but is included here for completeness.
Nmax < src < Nmax+1	–	0	yes	T(Nmax), fx(XX)
		1	yes	fx(XX), error()
src ≥ Nmax+1	0	–	–	T(Nmax), fx(VXCVI)
	1	–	–	fx(VXCVI), error()
src is a QNaN	0	–	–	T(Nmin), fx(VXCVI)
	1	–	–	fx(VXCVI), error()
src is a SNaN	0	–	–	T(Nmin), fx(VXCVI), fx(VXSNAN)
	1	–	–	fx(VXCVI), error(), fx(VXSNAN), error()

Explanation:

error()	The system error handler is invoked for the trap-enabled exception if MSR.FE0 and MSR.FE1 are set to any mode other than the ignore-exception mode. Update of VSR[XT] is suppressed.
f2i(x)	The double-precision floating-point integer value x is converted to 64-bit unsigned integer format.
fx(x)	FPSCR.FX is set to 1 if FPSCR.x=0. FPSCR.x is set to 1.
Nmin	The smallest unsigned integer doubleword value, 0 (0x0000_0000_0000_0000).
Nmax	The largest unsigned integer doubleword value, $2^{64} - 1$ (0xFFFF_FFFF_FFFF_FFFF).
src	The double-precision floating-point value in doubleword element i of VSR[XB] (where i∈{0,1}).
T(x)	The unsigned integer doubleword value x is placed in doubleword element i of VSR[XT] (where i∈{0,1}).
trunc(x)	The double-precision floating-point value x is truncated to a floating-point integer.

Table 7.86: Actions for xvcvdpuxds

VSX Vector Convert with round to zero Double-Precision to Unsigned Word format XX2-form

xvcvdpuxws XT,XB

60	T	///	B	200	BX	TX
0	6	11	16	21	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

ex_flag ← 0b0

do i = 0 to 1
   reset_xflags()

   src ← bfp_CONVERT_FROM_BFP64(VSR[32×BX+B].dword[i])
   rnd ← bfp_ROUND_TO_INTEGER(0b001, src)

   vresult.dword[i].word[0] ← ui32_CONVERT_FROM_BFP(rnd)
   vresult.dword[i].word[1] ← ui32_CONVERT_FROM_BFP(rnd)

   if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
   if vxcvi_flag=1 then SetFX(FPSCR.VXCVI)
   if xx_flag=1 then SetFX(FPSCR.XX)

   ex_flag ← ex_flag | (FPSCR.VE & vxsnan_flag)
                     | (FPSCR.VE & vxcvi_flag)
                     | (FPSCR.XE & xx_flag)
end

if ex_flag=0 then VSR[32×TX+T] ← vresult
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

For each integer value i from 0 to 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 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 result is also placed into bits 32:63 of doubleword element i of VSR[XT].

See Table 7.87.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered:

Register	*Field(s)*
FPSCR	FX XX VXSNAN VXCVI

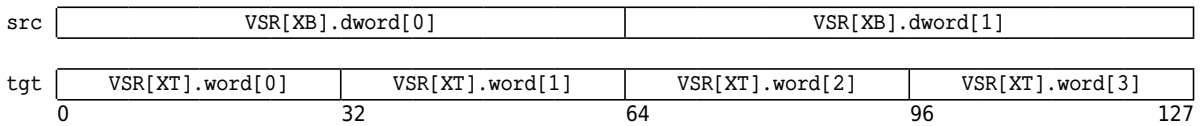
Programming Note

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.

Programming Note

Previous versions of the architecture allowed the contents of words 1 and 3 of the result register to be undefined. However, all processors that support this instruction write the result into words 0 and 1 and words 2 and 3 of the result register, as is required by this version of the architecture.

VSR Data Layout for xvcvdpuxws



	FPSCR.VE	FPSCR.XE	Inexact? (trunc(src) ≠ src)	Returned Results and Status Setting
src ≤ Nmin−1	0	–	–	T(Nmin), fx(VXCVI)
	1	–	–	fx(VXCVI), error()
Nmin−1 < src < Nmin	–	0	yes	T(Nmin), fx(XX)
		1	yes	fx(XX), error()
src = Nmin	–	–	no	T(Nmin)
Nmin < src < Nmax	–	–	no	T(f2i(trunc(src)))
		0	yes	T(f2i(trunc(src))), fx(XX)
		1	yes	fx(XX), error()
src = Nmax	–	–	no	T(Nmax)
Nmax < src < Nmax+1	–	0	yes	T(Nmax), fx(XX)
		1	yes	fx(XX), error()
src ≥ Nmax+1	0	–	–	T(Nmax), fx(VXCVI)
	1	–	–	fx(VXCVI), error()
src is a QNaN	0	–	–	T(Nmin), fx(VXCVI)
	1	–	–	fx(VXCVI), error()
src is a SNaN	0	–	–	T(Nmin), fx(VXCVI), fx(VXSNAN)
	1	–	–	fx(VXCVI), error(), fx(VXSNAN)

Explanation:

error()	The system error handler is invoked for the trap-enabled exception if MSR.FE0 and MSR.FE1 are set to any mode other than the ignore-exception mode. Update of VSR[XT] is suppressed.
f2i(x)	The double-precision floating-point integer value x is converted to 32-bit unsigned integer format.
fx(x)	FPSCR.FX is set to 1 if FPSCR.x=0. FPSCR.x is set to 1.
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 elements 2×i and 2×i+1 of VSR[XT] (where i∈{0,1}).
trunc(x)	The double-precision floating-point value x is truncated to a floating-point integer.

Table 7.87: Actions for xvcvdpuxws

VSX Vector Convert with round to zero Single-Precision to Signed Doubleword format XX2-form

xvcvspsxds XT,XB

60	T	///	B	408	BX	TX
0	6	11	16	21	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

ex_flag ← 0b0

do i = 0 to 1
   reset_xflags()

   src ← bfp_CONVERT_FROM_BFP32(
VSR[32×BX+B].dword[i].word[0])
   rnd ← bfp_ROUND_TO_INTEGER(0b001, src)

   vresult.dword[i] ← si64_CONVERT_FROM_BFP(rnd)

   if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
   if vxcvi_flag=1 then SetFX(FPSCR.VXCVI)
   if xx_flag=1 then SetFX(FPSCR.XX)

   ex_flag ← ex_flag | (FPSCR.VE & vxsnan_flag)
                     | (FPSCR.VE & vxcvi_flag)
                     | (FPSCR.XE & xx_flag)
end

if ex_flag=0 then VSR[32×TX+T] ← vresult
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

For each integer value i from 0 to 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 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 $2^{63}-1$, the result is 0x7FFF_FFFF_FFFF_FFFF and VXCVI is set to 1.

Otherwise, if the rounded value is less than -2^{63}, the result is 0x8000_0000_0000_0000 and VXCVI is set to 1.

Otherwise, the result is the rounded value converted to 64-bit signed-integer format, and if the result is inexact (i.e., not equal to src), XX is set to 1.

The result is placed into doubleword element i of VSR[XT].

See Table 7.88.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

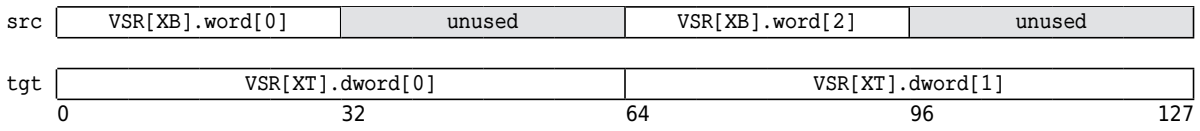
Special Registers Altered:

Register	*Field(s)*
FPSCR	FX XX VXSNAN VXCVI

Programming Note

xvcvspsxds 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 xvcvspsxds



	FPSCR.VE	FPSCR.XE	Inexact? (trunc(src) ≠ src)	Returned Results and Status Setting
src ≤ Nmin−1	0	–	–	T(Nmin), fx(VXCVI)
	1	–	–	fx(VXCVI), error()
Nmin−1 < src < Nmin	–	0	yes	T(Nmin), fx(XX)
		1	yes	fx(XX), error()
src = Nmin	–	–	no	T(Nmin)
Nmin < src < Nmax	–	–	no	T(f2i(trunc(src)))
		0	yes	T(f2i(trunc(src))), fx(XX)
		1	yes	fx(XX), error()
src = Nmax	–	–	no	T(Nmax) Note: This case cannot occur as Nmax is not representable in SP format but is included here for completeness.
Nmax < src < Nmax+1	–	0	yes	T(Nmax), fx(XX)
		1	yes	fx(XX), error()
src ≥ Nmax+1	0	–	–	T(Nmax), fx(VXCVI)
	1	–	–	fx(VXCVI), error()
src is a QNaN	0	–	–	T(Nmin), fx(VXCVI)
	1	–	–	fx(VXCVI), error()
src is a SNaN	0	–	–	T(Nmin), fx(VXCVI), fx(VXSNAN)
	1	–	–	fx(VXCVI), error(), fx(VXSNAN)

Explanation:

error()	The system error handler is invoked for the trap-enabled exception if MSR.FE0 and MSR.FE1 are set to any mode other than the ignore-exception mode. Update of VSR[XT] is suppressed.
f2i(x)	The single-precision floating-point integer value x is converted to 64-bit signed integer format.
fx(x)	FPSCR.FX is set to 1 if FPSCR.x=0. FPSCR.x is set to 1.
Nmin	The smallest signed integer doubleword value, -2^{63} (0x8000_0000_0000_0000).
Nmax	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∈{0,2}).
T(x)	The signed integer doubleword value x is placed in doubleword element i of VSR[XT] (where i∈{0,1}).
trunc(x)	The single-precision floating-point value x is truncated to a floating-point integer.

Table 7.88: Actions for xvcvspsxds

VSX Vector Convert with round to zero Single-Precision to Signed Word format XX2-form

xvcvspsxws XT,XB

60	T	///	B	152	BX	TX
0	6	11	16	21	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

ex_flag ← 0b0

do i = 0 to 3
   reset_xflags()

   src ← bfp_CONVERT_FROM_BFP32(VSR[32×BX+B].word[i])
   rnd ← bfp_ROUND_TO_INTEGER(0b001, src)

   vresult.word[i] ← si32_CONVERT_FROM_BFP(rnd)

   if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
   if vxcvi_flag=1 then SetFX(FPSCR.VXCVI)
   if xx_flag=1 then SetFX(FPSCR.XX)

   ex_flag ← ex_flag | (FPSCR.VE & vxsnan_flag)
                     | (FPSCR.VE & vxcvi_flag)
                     | (FPSCR.XE & xx_flag)
end

if ex_flag=0 then VSR[32×TX+T] ← vresult
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

For each integer value i from 0 to 3, do the following.

Let src be the 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 7.89.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

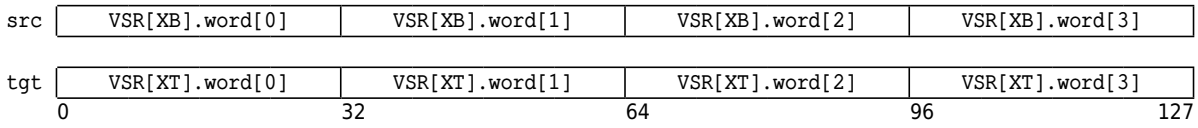
Special Registers Altered:

Register	*Field(s)*
FPSCR	FX XX VXSNAN VXCVI

Programming Note

xvcvspsxws 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 xvcvspsxws



	FPSCR.VE	FPSCR.XE	Inexact? (trunc(src) ≠ src)	Returned Results and Status Setting
src ≤ Nmin−1	0	–	–	T(Nmin), fx(VXCVI)
	1	–	–	fx(VXCVI), error()
Nmin−1 < src < Nmin	–	0	yes	T(Nmin), fx(XX)
		1	yes	fx(XX), error()
src = Nmin	–	–	no	T(Nmin)
Nmin < src < Nmax	–	–	no	T(f2i(trunc(src)))
		0	yes	T(f2i(trunc(src))), fx(XX)
		1	yes	fx(XX), error()
src = Nmax	–	–	no	T(Nmax) Note: This case cannot occur as Nmax is not representable in SP format but is included here for completeness.
Nmax < src < Nmax+1	–	0	yes	T(Nmax), fx(XX)
		1	yes	fx(XX), error()
src ≥ Nmax+1	0	–	–	T(Nmax), fx(VXCVI)
	1	–	–	fx(VXCVI), error()
src is a QNaN	0	–	–	T(Nmin), fx(VXCVI)
	1	–	–	fx(VXCVI), error()
src is a SNaN	0	–	–	T(Nmin), fx(VXCVI), fx(VXSNAN)
	1	–	–	fx(VXCVI), error(), fx(VXSNAN), error()

Explanation:

error()	The system error handler is invoked for the trap-enabled exception if MSR.FE0 and MSR.FE1 are set to any mode other than the ignore-exception mode. Update of VSR[XT] is suppressed.
f2i(x)	The single-precision floating-point integer value x is converted to 32-bit signed integer format.
fx(x)	FPSCR.FX is set to 1 if FPSCR.x=0. FPSCR.x is set to 1.
Nmin	The smallest signed integer word value, -2^{31} (0x8000_0000).
Nmax	The largest signed integer word value, $2^{31}-1$ (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}).
trunc(x)	The single-precision floating-point value x is truncated to a floating-point integer.

Table 7.89: Actions for xvcvspsxws

VSX Vector Convert with round to zero Single-Precision to Unsigned Doubleword format XX2-form

xvcvspuxds XT,XB

60	T	///	B	392	BX	TX
0	6	11	16	21	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

ex_flag ← 0b0

do i = 0 to 1
   reset_xflags()

   src ← bfp_CONVERT_FROM_BFP32(
            VSR[32×BX+B].dword[i].word[0])
   rnd ← bfp_ROUND_TO_INTEGER(0b001, src)

   vresult.dword[i] ← ui64_CONVERT_FROM_BFP(rnd)

   if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
   if vxcvi_flag=1 then SetFX(FPSCR.VXCVI)
   if xx_flag=1 then SetFX(FPSCR.XX)

   ex_flag ← ex_flag | (FPSCR.VE & vxsnan_flag)
                     | (FPSCR.VE & vxcvi_flag)
                     | (FPSCR.XE & xx_flag)
end

if ex_flag=0 then VSR[32×TX+T] ← vresult
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

For each integer value i from 0 to 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 7.90.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

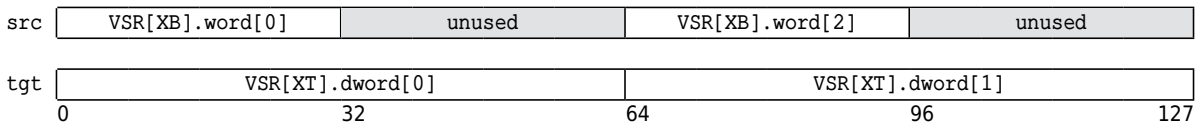
Special Registers Altered:

Register	*Field(s)*
FPSCR	FX XX VXSNAN VXCVI

Programming Note

xvcvspuxds rounds using Round towards Zero rounding mode. For other rounding modes, software must use a *Round to Single-Precision Integer* instruction that corresponds to the desired rounding mode, including ***xvrspic*** which uses the rounding mode specified by RN.

VSR Data Layout for xvcvspuxds



	FPSCR.VE	FPSCR.XE	Inexact? (trunc(src) ≠ src)	Returned Results and Status Setting
src ≤ Nmin−1	0	–	–	T(Nmin), fx(VXCVI)
	1	–	–	fx(VXCVI), error()
Nmin−1 < src < Nmin	–	0	yes	T(Nmin), fx(XX)
		1	yes	fx(XX), error()
src = Nmin	–	–	no	T(Nmin)
Nmin < src < Nmax	–	–	no	T(f2i(trunc(src)))
		0	yes	T(f2i(trunc(src))), fx(XX)
		1	yes	fx(XX), error()
src = Nmax	–	–	no	T(Nmax) Note: This case cannot occur as Nmax is not representable in SP format but is included here for completeness.
Nmax < src < Nmax+1	–	0	yes	T(Nmax), fx(XX)
		1	yes	fx(XX), error()
src ≥ Nmax+1	0	–	–	T(Nmax), fx(VXCVI)
	1	–	–	fx(VXCVI), error()
src is a QNaN	0	–	–	T(Nmin), fx(VXCVI)
	1	–	–	fx(VXCVI), error()
src is a SNaN	0	–	–	T(Nmin), fx(VXCVI), fx(VXSNAN)
	1	–	–	fx(VXCVI), error(), fx(VXSNAN), error()

Explanation:

error()	The system error handler is invoked for the trap-enabled exception if MSR.FE0 and MSR.FE1 are set to any mode other than the ignore-exception mode. Update of VSR[XT] is suppressed.
f2i(x)	The single-precision floating-point integer value x is converted to 64-bit unsigned integer format.
fx(x)	FPSCR.FX is set to 1 if FPSCR.x=0. FPSCR.x is set to 1.
Nmin	The smallest unsigned integer doubleword value, 0 (0x0000_0000_0000_0000).
Nmax	The largest unsigned integer doubleword value, $2^{64} - 1$ (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}).
trunc(x)	The single-precision floating-point value x is truncated to a floating-point integer.

Table 7.90: Actions for xvcvspuxds

VSX Vector Convert with round to zero Single-Precision to Unsigned Word format XX2-form

xvcvspuxws XT,XB

60	T	///	B	136	BX	TX
0	6	11	16	21	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

ex_flag ← 0b0

do i = 0 to 3
   reset_xflags()

   src ← bfp_CONVERT_FROM_BFP32(VSR[32×BX+B].word[i])
   rnd ← bfp_ROUND_TO_INTEGER(0b001, src)

   vresult.word[i] ← ui32_CONVERT_FROM_BFP(rnd)

   if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
   if vxcvi_flag=1 then SetFX(FPSCR.VXCVI)
   if xx_flag=1 then SetFX(FPSCR.XX)

   ex_flag ← ex_flag | (FPSCR.VE & vxsnan_flag)
                     | (FPSCR.VE & vxcvi_flag)
                     | (FPSCR.XE & xx_flag)
end

if ex_flag=0 then VSR[32×TX+T] ← vresult
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

For each integer value i from 0 to 3, do the following.

Let src be the 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 7.91.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

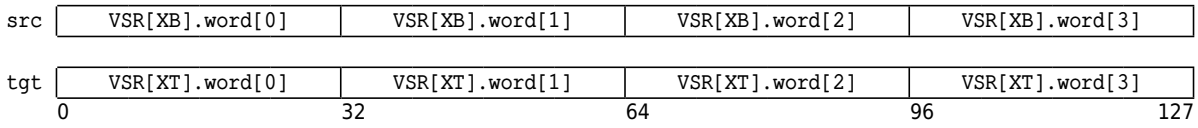
Special Registers Altered:

Register	*Field(s)*
FPSCR	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



	FPSCR.VE	FPSCR.XE	Inexact? (trunc(src) ≠ src)	Returned Results and Status Setting
src ≤ Nmin−1	0	–	–	T(Nmin), fx(VXCVI)
	1	–	–	fx(VXCVI), error()
Nmin−1 < src < Nmin	–	0	yes	T(Nmin), fx(XX)
		1	yes	fx(XX), error()
src = Nmin	–	–	no	T(Nmin)
Nmin < src < Nmax	–	–	no	T(f2i(trunc(src)))
		0	yes	T(f2i(trunc(src))), fx(XX)
		1	yes	fx(XX), error()
src = Nmax	–	–	no	T(Nmax) Note: This case cannot occur as Nmax is not representable in SP format but is included here for completeness.
Nmax < src < Nmax+1	–	0	yes	T(Nmax), fx(XX)
		1	yes	fx(XX), error()
src ≥ Nmax+1	0	–	–	T(Nmax), fx(VXCVI)
	1	–	–	fx(VXCVI), error()
src is a QNaN	0	–	–	T(Nmin), fx(VXCVI)
	1	–	–	fx(VXCVI), error()
src is a SNaN	0	–	–	T(Nmin), fx(VXCVI), fx(VXSNAN)
	1	–	–	fx(VXCVI), error(), fx(VXSNAN)

Explanation:

error()	The system error handler is invoked for the trap-enabled exception if MSR.FE0 and MSR.FE1 are set to any mode other than the ignore-exception mode. Update of VSR[XT] is suppressed.
f2i(x)	The single-precision floating-point integer value x is converted to 32-bit unsigned integer format.
fx(x)	FPSCR.FX is set to 1 if FPSCR.x=0. FPSCR.x is set to 1.
Nmin	The smallest unsigned integer word value, 0 (0x0000_0000).
Nmax	The largest unsigned integer word value, $2^{32} - 1$ (0xFFFF_FFFF).
src	The single-precision floating-point value in word element i of VSR[XB] (where i∈{0,1,2,3}).
T(x)	The unsigned integer word value x is placed in word element i of VSR[XT] (where i∈{0,1,2,3}).
trunc(x)	The single-precision floating-point value x is truncated to a floating-point integer.

Table 7.91: Actions for xvcvspuxws

7.6.2.10 VSX Binary Floating-Point Convert From Integer Instructions

7.6.2.10.1 VSX Scalar BFP Convert From Integer Instructions

VSX Scalar Convert Signed Doubleword to Quad-Precision format X-form

xscvsdqp VRT,VRB

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      ← bfp128_CONVERT_FROM_BFP(src)

VSR[VRT+32] ← result
FPSCR.FPRF  ← fprf_CLASS_BFP128(result)
FPSCR.FR    ← 0
FPSCR.FI    ← 0
```

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.

FPRF is set to the class and sign of the result. FR is set to 0. FI is set to 0.

Special Registers Altered:

Register	*Field(s)*
FPSCR	FPRF
FPSCR	FR (set to 0)
FPSCR	FI (set to 0)

VSX Scalar Convert Unsigned Doubleword to Quad-Precision format X-form

xscvudqp VRT,VRB

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      ← bfp128_CONVERT_FROM_BFP(src)

VSR[VRT+32] ← result
FPSCR.FPRF  ← fprf_CLASS_BFP128(result)
FPSCR.FR    ← 0
FPSCR.FI    ← 0
```

Let src be the unsigned integer value in doubleword element 0 of VSR[VRB+32].

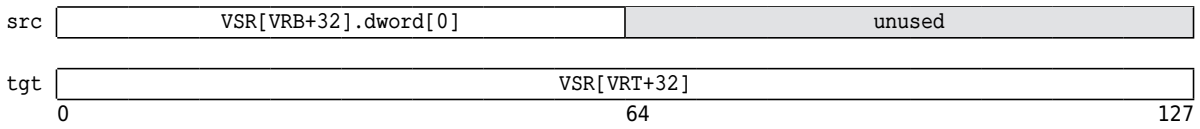
src is placed into VSR[VRT+32] in quad-precision floating-point format.

FPRF is set to the class and sign of the result. FR is set to 0. FI is set to 0.

Special Registers Altered:

Register	*Field(s)*
FPSCR	FPRF
FPSCR	FR (set to 0)
FPSCR	FI (set to 0)

VSR Data Layout for xscvsdqp & xscvudqp



VSX Scalar Convert with round Signed Quadword to Quad-Precision X-form

xscvsqqp VRT,VRB

63	VRT	11	VRB	836	/
0	6	11	16	21	31

```
if MSR.VSX=0 then VSX_Unavailable()

reset_xflags()

src    ← bfp_CONVERT_FROM_SI128(VSR[VRB+32])
rnd    ← bfp_ROUND_TO_BFP128(0,FPSCR.RN,src)
result ← bfp128_CONVERT_FROM_BFP(rnd)

if xx_flag=1 then SetFX(XX)

VSR[VRT+32] ← result
FPSCR.FPRF  ← fprf_CLASS_BFP128(result)
FPSCR.FR    ← inc_flag
FPSCR.FI    ← xx_flag
```

Let src be the 128-bit signed integer value in VSR[VRB+32].

src is converted to an unbounded-precision floating-point value and rounded to quad-precision using the rounding mode specified by RN.

The result is placed into VSR[VRT+32] in quad-precision format.

FPRF is set to the class and sign of the result. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.

Special Registers Altered:

Register	*Field(s)*
FPSCR	FPRF FR FI FX XX

VSX Scalar Convert with round Unsigned Quadword to Quad-Precision format X-form

xscvuqqp VRT,VRB

63	VRT	3	VRB	836	/
0	6	11	16	21	31

```
if MSR.VSX=0 then VSX_Unavailable()

reset_xflags()

src    ← bfp_CONVERT_FROM_UI128(VSR[VRB+32])
rnd    ← bfp_ROUND_TO_BFP128(0b0,FPSCR.RN,src)
result ← bfp128_CONVERT_FROM_BFP(rnd)

if xx_flag=1 then SetFX(XX)

VSR[VRT+32] ← result
FPSCR.FPRF  ← fprf_CLASS_BFP128(result)
FPSCR.FR    ← inc_flag
FPSCR.FI    ← xx_flag
```

Let src be the 128-bit unsigned integer value in VSR[VRB+32].

src is converted to an unbounded-precision floating-point value and rounded to quad-precision using the rounding mode specified by RN.

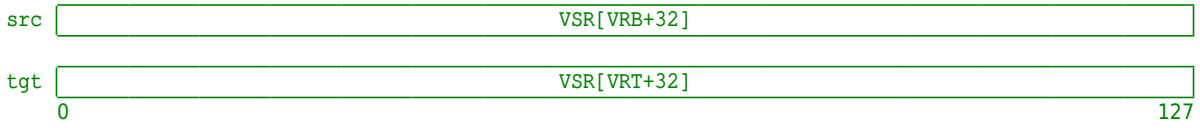
The result is placed into VSR[VRT+32] in quad-precision format.

FPRF is set to the class and sign of the result. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.

Special Registers Altered:

Register	*Field(s)*
FPSCR	FPRF FR FI FX XX

VSR Data Layout for xscvsqqp & xscvuqqp



VSX Scalar Convert with round Signed Doubleword to Double-Precision format XX2-form

xscvsxddp XT,XB

60	T	///	B	376	BX	TX
0	6	11	16	21	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

reset_xflags()

src    ← bfp_CONVERT_FROM_SI64(VSR[32×BX+B].dword[0])
rnd    ← bfp_ROUND_TO_BFP64(0b0,FPSCR.RN,v)
result ← bfp64_CONVERT_FROM_BFP(rnd)

if xx_flag=1 then SetFX(FPSCR.XX)

VSR[32×TX+T].dword[0] ← result
VSR[32×TX+T].dword[1] ← 0x0000_0000_0000_0000
FPSCR.FPRF ← fprf_CLASS_BFP64(result)
FPSCR.FR   ← inc_flag
FPSCR.FI   ← xx_flag
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

Let src be the signed integer value in doubleword element 0 of VSR[XB].

src is converted to an unbounded-precision floating-point value and rounded to double-precision using the rounding mode specified by RN.

The result is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are set to 0.

FPRF is set to the class and sign of the result. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.

Special Registers Altered:

Register	*Field(s)*
FPSCR	FPRF FR FI FX XX

Programming Note

Previous versions of the architecture allowed the contents of doubleword 1 of the result register to be undefined. However, all processors that support this instruction write 0s into doubleword 1 of the result register, as is required by this version of the architecture.

VSX Scalar Convert with round Unsigned Doubleword to Double-Precision format XX2-form

xscvuxddp XT,XB

60	T	///	B	360	BX	TX
0	6	11	16	21	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

reset_xflags()

src    ← bfp_CONVERT_FROM_UI64(VSR[32×BX+B].dword[0])
rnd    ← bfp_ROUND_TO_BFP64(0b0,FPSCR.RN,v)
result ← bfp64_CONVERT_FROM_BFP(rnd)

if xx_flag=1 then SetFX(FPSCR.XX)

VSR[32×TX+T].dword[0] ← result
VSR[32×TX+T].dword[1] ← 0x0000_0000_0000_0000
FPSCR.FPRF ← fprf_CLASS_BFP64(result)
FPSCR.FR   ← inc_flag
FPSCR.FI   ← xx_flag
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

Let src be the unsigned integer value in doubleword element 0 of VSR[XB].

src is converted to an unbounded-precision floating-point value and rounded to double-precision using the rounding mode specified by RN.

The result is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are set to 0.

FPRF is set to the class and sign of the result. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.

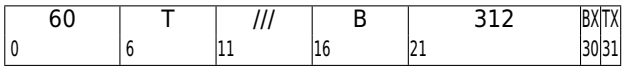
Special Registers Altered:

Register	*Field(s)*
FPSCR	FPRF FR FI FX XX

VSR Data Layout for xscvsxddp & xscvuxddp

VSX Scalar Convert with round Signed Doubleword to Single-Precision format XX2-form

xscvsxdsp XT,XB




```
if MSR.VSX=0 then VSX_Unavailable()

reset_xflags()

src      ← bfp_CONVERT_FROM_SI64(VSR[32×BX+B].dword[0])
rnd      ← bfp_ROUND_TO_BFP32(FPSCR.RN,v)
result32 ← bfp32_CONVERT_FROM_BFP(rnd)
result64 ← bfp64_CONVERT_FROM_BFP(rnd)

if xx_flag=1 then SetFX(FPSCR.XX)

VSR[32×TX+T].dword[0] ← result64
VSR[32×TX+T].dword[1] ← 0x0000_0000_0000_0000
FPSCR.FPRF ← fprf_CLASS_BFP32(result32)
FPSCR.FR   ← inc_flag
FPSCR.FI   ← xx_flag
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

Let src be the two's-complement integer value in doubleword element 0 of VSR[XB].

src is converted to floating-point format, and rounded to single-precision using the rounding mode specified by RN.

The result is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are set to 0.

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:

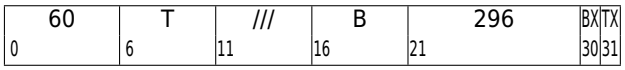
Register	*Field(s)*
FPSCR	FPRF FR FI FX XX

Programming Note

Previous versions of the architecture allowed the contents of doubleword 1 of the result register to be undefined. However, all processors that support this instruction write 0s into doubleword 1 of the result register, as is required by this version of the architecture.

VSX Scalar Convert with round Unsigned Doubleword to Single-Precision format XX2-form

xscvuxdsp XT,XB




```
if MSR.VSX=0 then VSX_Unavailable()

reset_xflags()

src      ← bfp_CONVERT_FROM_UI64(VSR[32×BX+B].dword[0])
rnd      ← bfp_ROUND_TO_BFP32(FPSCR.RN,v)
result32 ← bfp32_CONVERT_FROM_BFP(rnd)
result64 ← bfp64_CONVERT_FROM_BFP(rnd)

if xx_flag=1 then SetFX(FPSCR.XX)

VSR[32×TX+T].dword[0] ← result64
VSR[32×TX+T].dword[1] ← 0x0000_0000_0000_0000
FPSCR.FPRF ← fprf_CLASS_BFP32(result32)
FPSCR.FR   ← inc_flag
FPSCR.FI   ← xx_flag
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

Let src be the unsigned-integer value in doubleword element 0 of VSR[XB].

src is converted to floating-point format, and rounded to single-precision using the rounding mode specified by RN.

The result is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are set to 0.

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:

Register	*Field(s)*
FPSCR	FPRF FR FI FX XX

Programming Note

Previous versions of the architecture allowed the contents of doubleword 1 of the result register to be undefined. However, all processors that support this instruction write 0s into doubleword 1 of the result register, as is required by this version of the architecture.

VSR Data Layout for xscvsxdsp & xscvuxdsp

7.6.2.10.2 VSX Vector BFP Convert From Integer Instructions

VSX Vector Convert with round Signed Doubleword to Double-Precision format XX2-form

xvcvsxddp XT,XB

60	T	///	B	504	BX	TX
0	6	11	16	21	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

ex_flag ← 0b0

do i = 0 to 1
   reset_xflags()

   src ← bfp_CONVERT_FROM_SI64(VSR[32×BX+B].dword[i])
   rnd ← bfp_ROUND_TO_BFP64(0b0,FPSCR.RN,v)

   vresult.dword[i] ← bfp64_CONVERT_FROM_BFP(rnd)

   if xx_flag=1 then SetFX(FPSCR.XX)

   ex_flag ← ex_flag | (FPSCR.XE & xx_flag)
end

if ex_flag=0 then VSR[32×TX+T] ← vresult
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

For each integer value i from 0 to 1, do the following.

Let src be the signed integer in doubleword element i of VSR[XB].

src is converted to an unbounded-precision floating-point value and rounded to 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:

Register	*Field(s)*
FPSCR	FX XX

VSX Vector Convert with round Unsigned Doubleword to Double-Precision format XX2-form

xvcvuxddp XT,XB

60	T	///	B	488	BX	TX
0	6	11	16	21	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

ex_flag ← 0b0

do i = 0 to 1
   reset_xflags()

   src ← bfp_CONVERT_FROM_UI64(VSR[32×BX+B].dword[i])
   rnd ← bfp_ROUND_TO_BFP64(0b0,FPSCR.RN,src)

   vresult.dword[i] ← bfp64_CONVERT_FROM_BFP(rnd)

   if xx_flag=1 then SetFX(FPSCR.XX)

   ex_flag ← ex_flag | (FPSCR.XE & xx_flag)
end

if ex_flag=0 then VSR[32×TX+T] ← vresult
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

For each integer value i from 0 to 1, do the following.

Let src be the unsigned integer in doubleword element i of VSR[XB].

src is converted to an unbounded-precision floating-point value and rounded to 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].

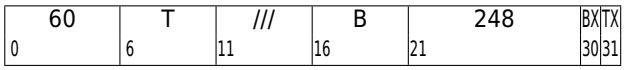
Special Registers Altered:

Register	*Field(s)*
FPSCR	FX XX

VSR Data Layout for xvcvsxddp & xvcvuxddp

VSX Vector Convert Signed Word to Double-Precision format XX2-form

xvcvsxwdp XT,XB




```
if MSR.VSX=0 then VSX_Unavailable()

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 integer value i from 0 to 1, do the following.

Let src be the signed integer value in bits 0:31 of doubleword element i of VSR[XB].

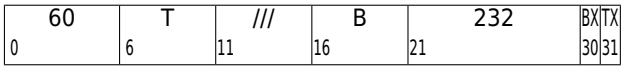
src is placed into doubleword element i of VSR[XT] in double-precision format.

Special Registers Altered:

None

VSX Vector Convert Unsigned Word to Double-Precision format XX2-form

xvcvuxwdp XT,XB




```
if MSR.VSX=0 then VSX_Unavailable()

do i = 0 to 1
   src ← bfp_CONVERT_FROM_UI32(
               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 integer value i from 0 to 1, do the following.

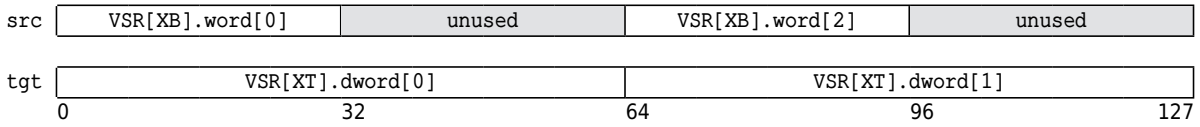
Let src be the unsigned integer value in bits 0:31 of doubleword element i of VSR[XB].

src is placed into doubleword element i of VSR[XT] in double-precision format.

Special Registers Altered:

None

VSR Data Layout for xvcvsxwdp & xvcvuxwdp



VSX Vector Convert with round Signed Doubleword to Single-Precision format XX2-form

xvcvsxdsp XT,XB

60	T	///	B	440	BX	TX
0	6	11	16	21	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

ex_flag ← 0b0

do i = 0 to 1
   reset_xflags()

   src ← bfp_CONVERT_FROM_SI64(VSR[32×BX+B].dword[i])
   rnd ← bfp_ROUND_TO_BFP32(FPSCR.RN,v)

   vresult.dword[i].word[0] ← bfp32_CONVERT_FROM_BFP(rnd)
   vresult.dword[i].word[1] ← bfp32_CONVERT_FROM_BFP(rnd)

   if xx_flag=1 then SetFX(FPSCR.XX)

   ex_flag ← ex_flag | (FPSCR.XE & xx_flag)
end

if ex_flag=0 then VSR[32×TX+T] ← vresult
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

For each integer value i from 0 to 1, do the following.

Let src be the signed integer in doubleword element i of VSR[XB].

src is converted to an unbounded-precision floating-point value and rounded to single-precision using the rounding mode specified by RN.

The result is placed into bits 0:31 of doubleword element i of VSR[XT] in single-precision format.

The result is also placed into bits 32:63 of doubleword element i of VSR[XT] in single-precision format.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered:

Register	*Field(s)*
FPSCR	FX XX

VSX Vector Convert with round Unsigned Doubleword to Single-Precision format XX2-form

xvcvuxdsp XT,XB

60	T	///	B	424	BX	TX
0	6	11	16	21	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

ex_flag ← 0b0

do i = 0 to 1
   reset_xflags()

   src ← bfp_CONVERT_FROM_UI64(VSR[32×BX+B].dword[i])
   rnd ← bfp_ROUND_TO_BFP32(FPSCR.RN,src)

   vresult.dword[i].word[0] ← bfp32_CONVERT_FROM_BFP(rnd)
   vresult.dword[i].word[1] ← bfp32_CONVERT_FROM_BFP(rnd)

   if xx_flag=1 then SetFX(FPSCR.XX)

   ex_flag ← ex_flag | (FPSCR.XE & xx_flag)
end

if ex_flag=0 then VSR[32×TX+T] ← vresult
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

For each integer value i from 0 to 1, do the following.

Let src be the unsigned integer in doubleword element i of VSR[XB].

src is converted to an unbounded-precision floating-point value and rounded to single-precision using the rounding mode specified by RN.

The result is placed into bits 0:31 of doubleword element i of VSR[XT] in single-precision format.

The result is also placed into bits 32:63 of doubleword element i of VSR[XT] in single-precision format.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

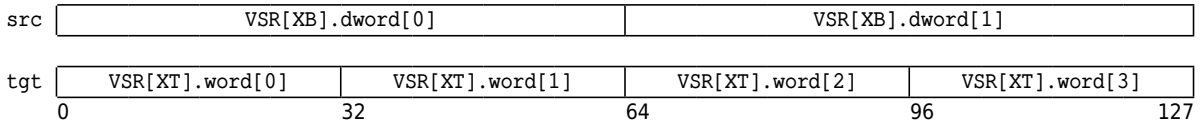
Special Registers Altered:

Register	*Field(s)*
FPSCR	FX XX

Programming Note

Previous versions of the architecture allowed the contents of words 1 and 3 of the result register to be undefined. However, all processors that support these instructions write the result into words 0 and 1 and words 2 and 3 of the result register, as is required by this version of the architecture.

VSR Data Layout for xvcvsxdsp & xvcvuxdsp



VSX Vector Convert with round Signed Word to Single-Precision format XX2-form

xvcvsxwsp XT,XB

60	T	///	B	184	BX	TX
0	6	11	16	21	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

ex_flag ← 0b0

do i = 0 to 3
   reset_xflags()

   src ← bfp_CONVERT_FROM_SI32(VSR[32×BX+B].word[i])
   rnd ← bfp_ROUND_TO_BFP32(FPSCR.RN,src)

   vresult.word[i] ← bfp32_CONVERT_FROM_BFP(rnd)

   if xx_flag=1 then SetFX(FPSCR.XX)
   ex_flag ← ex_flag | (FPSCR.XE & xx_flag)
end

if ex_flag=0 then VSR[32×TX+T] ← vresult
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

For each integer value i from 0 to 3, do the following.

Let src be the signed integer in word element i of VSR[XB].

src is converted to an unbounded-precision floating-point value and rounded to single-precision using the rounding mode specified by RN.

The result is placed into word element i of VSR[XT] in single-precision format.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered:

Register	*Field(s)*
FPSCR	FX XX

VSX Vector Convert with round Unsigned Word to Single-Precision format XX2-form

xvcvuxwsp XT,XB

60	T	///	B	168	BX	TX
0	6	11	16	21	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

ex_flag ← 0b0

do i = 0 to 3
   reset_xflags()

   src ← bfp_CONVERT_FROM_UI32(VSR[32×BX+B].word[i])
   rnd ← bfp_ROUND_TO_BFP32(FPSCR.RN,v)

   vresult.word[i] ← bfp32_CONVERT_FROM_BFP(rnd)

   if xx_flag=1 then SetFX(FPSCR.XX)
   ex_flag ← ex_flag | (FPSCR.XE & xx_flag)
end

if ex_flag=0 then VSR[32×TX+T] ← vresult
```

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 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.

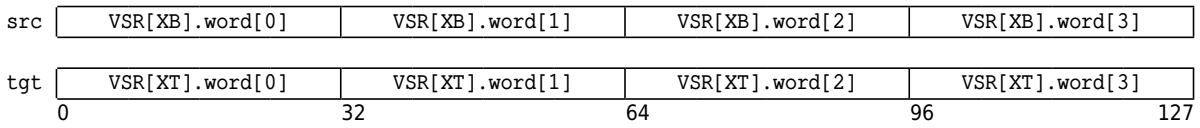
The result is placed into word element i of VSR[XT] in single-precision format.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered:

Register	*Field(s)*
FPSCR	FX XX

VSR Data Layout for xvcvsxwsp & xvcvuxwsp



7.6.2.11 VSX Binary Floating-Point Math Support Instructions

7.6.2.11.1 VSX Scalar BFP Math Support Instructions

VSX Scalar Compare Exponents Double-Precision XX3-form

xscmpexpdp BF,XA,XB

60	BF	//	A	B	59	AX	BX	/
0	6	9	11	16	21	29	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

src1 ← VSR[32×AX+A].dword[0]
src2 ← VSR[32×BX+B].dword[0]

src1.exponent ← EXTZ(src1.bit[1:11])
src2.exponent ← EXTZ(src2.bit[1:11])
src1.fraction ← EXTZ(src1.bit[12:63])
src2.fraction ← EXTZ(src2.bit[12:63])

src1.class.NaN ← (src1.exponent = 2047) &
                 (src1.fraction != 0)
src2.class.NaN ← (src2.exponent = 2047) &
                 (src2.fraction != 0)

lt_flag ← (src1.exponent < src2.exponent)
gt_flag ← (src1.exponent > src2.exponent)
eq_flag ← (src1.exponent = src2.exponent)
uo_flag ← src1.class.NaN | src2.class.NaN

CR.bit[4×BF+32] ← FPSCR.FL ← !uo_flag & lt_flag
CR.bit[4×BF+33] ← FPSCR.FG ← !uo_flag & gt_flag
CR.bit[4×BF+34] ← FPSCR.FE ← !uo_flag & eq_flag
CR.bit[4×BF+35] ← FPSCR.FU ← uo_flag
```

Let XA be the sum 32×AX + A.
Let XB be the sum 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].

The exponent of src1 is compared with the exponent of src2. The result of the compare is placed into FPCC and CR field BF.

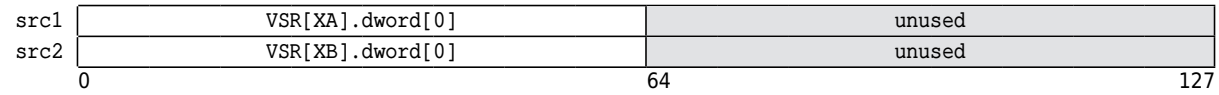
Special Registers Altered:

Register	*Field(s)*
CR	field BF
FPSCR	FPCC

Programming Note

This instruction can be used to operate on single-precision source operands.

VSR Data Layout for xscmpexpdp



VSX Scalar Compare Exponents Quad-Precision X-form

xscmpexpqp BF,VRA,VRB

63	BF	//	VRA	VRB	164	/
0	6	9	11	16	21	31

```
if MSR.VSX=0 then VSX_Unavailable()

reset_flags()

src1           ← VSR[VRA+32]
src2           ← VSR[VRB+32]

src1.exponent  ← EXTZ(src1.bit[1:15])
src2.exponent  ← EXTZ(src2.bit[1:15])
src1.fraction  ← EXTZ(src1.bit[16:127])
src2.fraction  ← EXTZ(src2.bit[16:127])

src1.class.NaN ← (src1.exponent = 32767) &
                 (src1.fraction != 0)
src2.class.NaN ← (src2.exponent = 32767) &
                 (src2.fraction != 0)

lt_flag ← (src1.exponent < src2.exponent)
gt_flag ← (src1.exponent > src2.exponent)
eq_flag ← (src1.exponent = src2.exponent)
uo_flag ← src1.class.NaN | src2.class.NaN

CR.bit[4×BF+32] ← FPSCR.FL ← !uo_flag & lt_flag
CR.bit[4×BF+33] ← FPSCR.FG ← !uo_flag & gt_flag
CR.bit[4×BF+34] ← FPSCR.FE ← !uo_flag & eq_flag
CR.bit[4×BF+35] ← FPSCR.FU ← uo_flag
```

Let `src1` be the floating-point value in `VSR[VRA+32]` represented in quad-precision format.

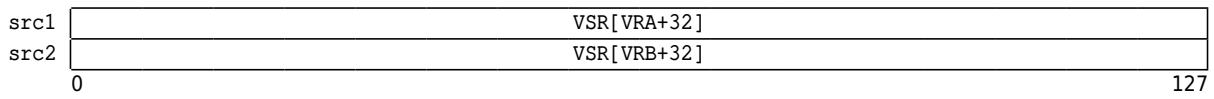
Let `src2` be the floating-point value in `VSR[VRB+32]` represented in quad-precision format.

The exponent of `src1` is compared with the exponent of `src2` as unsigned integer values. The result of the compare is placed into `FPCC` and `CR` field `BF`.

Special Registers Altered:

Register	*Field(s)*
CR	field BF
FPSCR	FPCC

VSR Data Layout for xscmpexpqp



VSX Scalar Insert Exponent Double-Precision X-form

xsiexpdp XT,RA,RB

60	T	RA	RB	918	TX
0	6	11	16	21	31

```
if MSR.VSX=0 then VSX_Unavailable()

src1 ← GPR[RA]
src2 ← GPR[RB]

XT ← 32×TX + T
VSR[XT].dword[0].bit[0]     ← src1.bit[0]
VSR[XT].dword[0].bit[1:11]  ← src2.bit[53:63]
VSR[XT].dword[0].bit[12:63] ← src1.bit[12:63]
VSR[XT].dword[1]            ← 0x0000_0000_0000_0000
```

Let XT be the sum 32×TX + T.

Let src1 be the unsigned integer value in GPR[RA].

Let src2 be the unsigned integer value in GPR[RB].

The contents of bit 0 of src1 are placed into bit 0 of VSR[XT].

The contents of bits 53:63 of src2 are placed into bits 1:11 of VSR[XT].

The contents of bits 12:63 of src1 are placed into bits 12:63 of VSR[XT].

The contents of doubleword element 1 of VSR[XT] are set to 0.

Special Registers Altered:

None

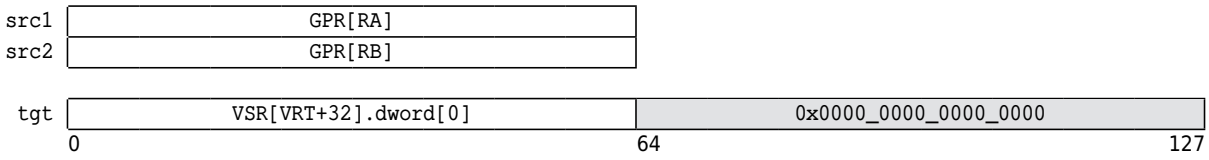
Programming Note

This instruction can be used to produce a single-precision result.

Programming Note

Previous versions of the architecture allowed the contents of doubleword 1 of the result register to be undefined. However, all processors that support this instruction write 0s into doubleword 1 of the result register, as is required by this version of the architecture.

VSR Data Layout for xsiexpdp



VSX Scalar Insert Exponent Quad-Precision X-form

xsiexpqp VRT,VRA,VRB

63	VRT	VRA	VRB	868	/
0	6	11	16	21	31

```
if MSR.VSX=0 then VSX_Unavailable()

VSR[VRT+32].bit[0]      ← VSR[VRA+32].bit[0]
VSR[VRT+32].bit[1:15]   ←
      VSR[VRB+32].dword[0].bit[49:63]
VSR[VRT+32].bit[16:127] ← VSR[VRA+32].bit[16:127]
```

The contents of bit 0 of `VSR[VRA+32]` are placed into bit 0 of `VSR[VRT+32]`.

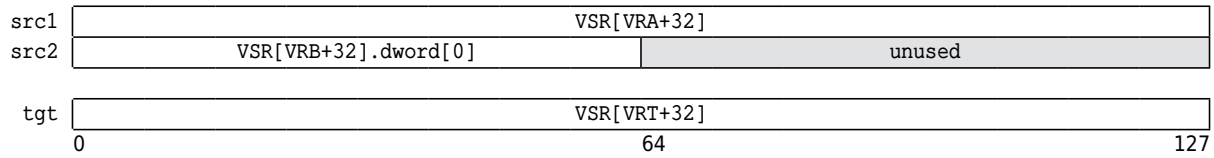
The contents of bit 49:63 of doubleword element 0 of `VSR[VRB+32]` are placed into bits 1:15 of `VSR[VRT+32]`.

The contents of bit 16:127 of `VSR[VRA+32]` are placed into bits 16:127 of `VSR[VRT+32]`.

Special Registers Altered:

None

VSR Data Layout for xsiexpqp



VSX Scalar Test Data Class Double-Precision XX2-form

xststdcdp BF,XB,DCMX

60	BF	DCMX	B	362	BX	/
0	6	9	16	21	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

src            ← VSR[32×BX+B].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)

match ←
  (DCMX.bit[0] & class.NaN)              |
  (DCMX.bit[1] & class.Infinity & !sign) |
  (DCMX.bit[2] & class.Infinity &  sign) |
  (DCMX.bit[3] & class.Zero     & !sign) |
  (DCMX.bit[4] & class.Zero     &  sign) |
  (DCMX.bit[5] & class.Denormal & !sign) |
  (DCMX.bit[6] & class.Denormal &  sign)

CR.bit[4×BF+32] ← FPSCR.FL ← src.sign
CR.bit[4×BF+33] ← FPSCR.FG ← 0b0
CR.bit[4×BF+34] ← FPSCR.FE ← match
CR.bit[4×BF+35] ← FPSCR.FU ← 0b0
```

Let XB be the sum 32×BX + B.

Let src be the double-precision floating-point value in doubleword element 0 of VSR[XB].

Bit 0 of CR field BF and bit 0 of FPCC are set to the sign bit of src.

Bit 1 of CR field BF and bit 1 of FPCC are set to 0b0.

Bit 2 of CR field BF and bit 2 of FPCC are set to indicate whether the data class of src, as represented in double-precision format, matches any of the data classes specified by DCMX (Data Class Mask).

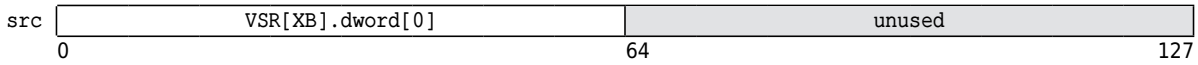
DCMX bit	Data Class
0	NaN
1	+Infinity
2	−Infinity
3	+Zero
4	−Zero
5	+Denormal
6	−Denormal

Bit 3 of CR field BF and bit 3 of FPCC are set to 0b0.

Special Registers Altered:

Register	*Field(s)*
CR	field BF
FPSCR	FPCC

VSR Data Layout for xststdcdp



VSX Scalar Test Data Class Quad-Precision X-form

xststdcqp BF,VRB,DCMX

63	BF	DCMX	VRB	708	/
0	6	9	16	21	31

```
if MSR.VSX=0 then VSX_Unavailable()

src            ← VSR[VRB+32]
exponent       ← src.bit[1:15]
fraction       ← src.bit[16:127]

class.Infinity ← (exponent = 0x7FFF) & (fraction  = 0)
class.NaN      ← (exponent = 0x7FFF) & (fraction != 0)
class.Zero     ← (exponent = 0x0000) & (fraction  = 0)
class.Denormal ← (exponent = 0x0000) & (fraction != 0)

match ←
  (DCMX.bit[0] & class.NaN)              |
  (DCMX.bit[1] & class.Infinity & !sign) |
  (DCMX.bit[2] & class.Infinity &  sign) |
  (DCMX.bit[3] & class.Zero     & !sign) |
  (DCMX.bit[4] & class.Zero     &  sign) |
  (DCMX.bit[5] & class.Denormal & !sign) |
  (DCMX.bit[6] & class.Denormal &  sign)

CR.bit[4×BF+32] ← FPSCR.FL ← src.sign
CR.bit[4×BF+33] ← FPSCR.FG ← 0b0
CR.bit[4×BF+34] ← FPSCR.FE ← match
CR.bit[4×BF+35] ← FPSCR.FU ← 0b0
```

Let 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.

DCMX bit	Data Class
0	NaN
1	+Infinity
2	−Infinity
3	+Zero
4	−Zero
5	+Denormal
6	−Denormal

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.

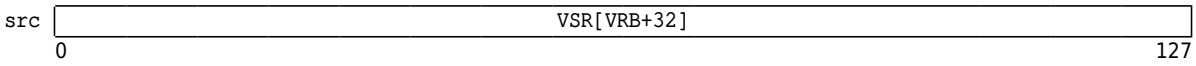
Bit 2 of CR field BF and bit 2 of FPCC are set to indicate whether the data class of src, as represented in quad-precision format, matches any of the data classes specified by DCM.

Bit 3 of CR field BF and bit 3 of FPCC are set to 0b0.

Special Registers Altered:

Register	*Field(s)*
CR	field BF
FPSCR	FPCC

VSR Data Layout for xststdcqp



VSX Scalar Test Data Class Single-Precision XX2-form

xststdcsp BF,XB,DCMX

60	BF	DCMX	B	298	BX	/
0	6	9	16	21	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

src            ← VSR[32×BX+B].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)
    | (exponent > 0x000) & (exponent < 0x381)

match ←
  (DCMX.bit[0] & class.NaN)              |
  (DCMX.bit[1] & class.Infinity & !sign) |
  (DCMX.bit[2] & class.Infinity &  sign) |
  (DCMX.bit[3] & class.Zero     & !sign) |
  (DCMX.bit[4] & class.Zero     &  sign) |
  (DCMX.bit[5] & class.Denormal & !sign) |
  (DCMX.bit[6] & class.Denormal &  sign)

not_SP_value ← ¬bfp64_IS_BFP32_VALUE(src)

CR.bit[4×BF]   ← FPSCR.FL ← src.sign
CR.bit[4×BF+1] ← FPSCR.FG ← 0b0
CR.bit[4×BF+2] ← FPSCR.FE ← match
CR.bit[4×BF+3] ← FPSCR.FU ← not_SP_value
```

Let XB be the sum 32×BX + B.

Let src be the double-precision floating-point value in doubleword element 0 of VSR[XB].

Bit 0 of CR field BF and bit 0 of FPCC are set to the sign bit of src.

Bit 1 of CR field BF and bit 1 of FPCC are set to 0b0.

Bit 2 of CR field BF and bit 2 of FPCC are set to indicate whether the data class of src, as represented in single-precision format, matches any of the data classes specified by DCMX (Data Class Mask).

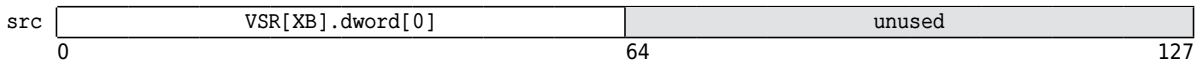
DCMX bit	Data Class
0	NaN
1	+Infinity
2	−Infinity
3	+Zero
4	−Zero
5	+Denormal
6	−Denormal

Bit 3 of CR field BF and bit 3 of FPCC are set to indicate if src is not representable in single-precision format.

Special Registers Altered:

Register	*Field(s)*
CR	field BF
FPSCR	FPCC

VSR Data Layout for xststdcsp



VSX Scalar Extract Exponent Double-Precision XX2-form

xsxexpdp RT,XB

60	RT	0	B	347	BX	/
0	6	11	16	21	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

src ← VSR[32×BX+B].dword[0]

GPR[RT] ← EXTZ64(src.bit[1:11])
```

Let XB be the sum 32×BX + B.

Let src be the double-precision floating-point value in doubleword element 0 of VSR[XB].

The value of the exponent field in src is placed into GPR[RT] in unsigned integer format.

Special Registers Altered:

None

Programming Note

This instruction can be used to operate on a single-precision source operand.

VSX Scalar Extract Exponent Quad-Precision X-form

xsxexpqp VRT,VRB

63	VRT	2	VRB	804	/
0	6	11	16	21	31

```
if MSR.VSX=0 then VSX_Unavailable()

src ← VSR[VRB+32]
VSR[VRT+32].dword[0] ← EXTZ64(src.bit[1:15]))
VSR[VRT+32].dword[1] ← 0x0000_0000_0000_0000
```

Let src be the quad-precision floating-point value in VSR[VRB+32].

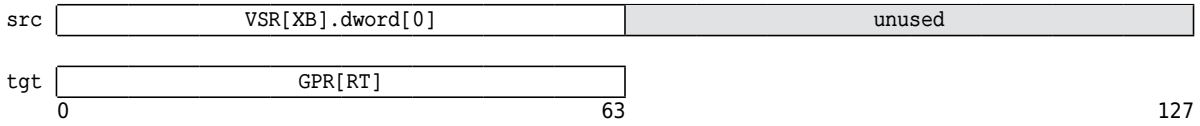
The contents of the exponent field of src (bits 1:15) are zero-extended and placed into doubleword 0 of VSR[VRT+32].

The contents of doubleword 1 of VSR[VRT+32] are set to 0.

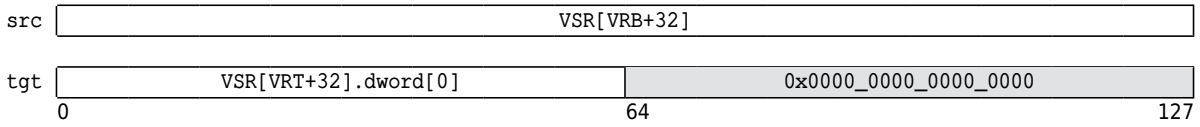
Special Registers Altered:

None

VSR Data Layout for xsxexpdp



VSR Data Layout for xsxexqp



VSX Scalar Extract Significand Double-Precision XX2-form

xsxsigdp RT,XB

60	RT	1	B	347	BX	/
0	6	11	16	21	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

exponent ← VSR[32×BX+B].bit[1:11]
fraction ← EXTZ64(VSR[32×BX+B].bit[12:63])

if (exponent != 0) & (exponent != 2047) then
   significand ← fraction | (0x001 || 52 0)
else
   significand ← fraction

GPR[RT] ← significand
```

Let XB be the sum 32×BX + B.

Let src be the double-precision floating-point value in doubleword element 0 of VSR[XB].

The significand of src is placed into GPR[RT] in unsigned integer format. If src is a normal value, the implicit leading bit is set to 1.

Special Registers Altered:

None

Programming Note

This instruction can be used to operate on a single-precision source operand.

VSX Scalar Extract Significand Quad-Precision X-form

xsxsigqp VRT,VRB

63	VRT	18	VRB	804	/
0	6	11	16	21	31

```
if MSR.VSX=0 then VSX_Unavailable()

src      ← VSR[VRB+32]
exponent ← EXTZ(src.bit[1:15])
fraction ← EXTZ128(src.bit[16:127])

if (exponent != 0) &  (exponent != 32767) then
   VSR[VRT+32] ← fraction | (0x0001 || 112 0)
else
   VSR[VRT+32] ← fraction
```

Let src be the quad-precision floating-point value in VSR[VRB+32].

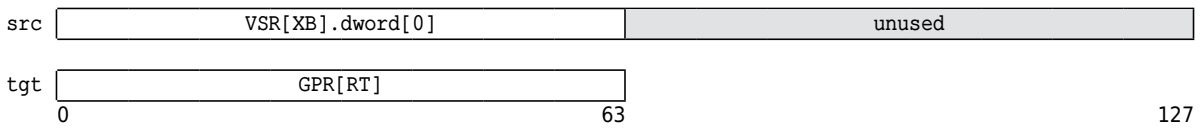
The significand of src is placed into VSR[VRT+32].

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_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.

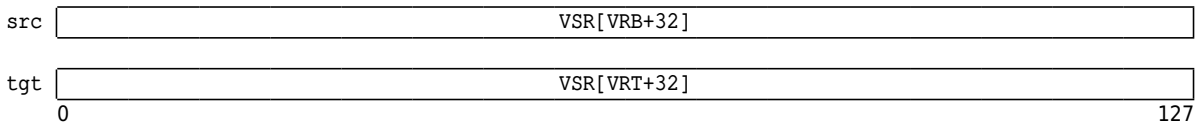
Special Registers Altered:

None

VSR Data Layout for xsxsigdp



VSR Data Layout for xsxsigqp



7.6.2.11.2 VSX Vector BFP Math Support Instructions

VSX Vector Insert Exponent Double-Precision XX3-form

xviexpdp XT,XA,XB

60	T	A	B	248	AX	BX	TX
0	6	11	16	21	29	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

XT ← 32×TX + T
do i = 0 to 1
   src1 ← VSR[32×AX+A].dword[i]
   src2 ← VSR[32×BX+B].dword[i]

   VSR[XT].dword[i].bit[0]     ← src1.bit[0]
   VSR[XT].dword[i].bit[1:11]  ← src2.bit[53:63]
   VSR[XT].dword[i].bit[12:63] ← src1.bit[12:63]
end
```

Let XT be the sum 32×TX + T.
Let XA be the sum 32×AX + A.
Let XB be the sum 32×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

VSX Vector Insert Exponent Single-Precision XX3-form

xviexpsp XT,XA,XB

60	T	A	B	216	AX	BX	TX
0	6	11	16	21	29	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

XT ← 32×TX + T
do i = 0 to 3
   src1 ← VSR[32×AX+A].word[i]
   src2 ← VSR[32×BX+B].word[i]

   VSR[XT].word[i].bit[0]    ← src1.bit[0]
   VSR[XT].word[i].bit[1:8]  ← src2.bit[24:31]
   VSR[XT].word[i].bit[9:31] ← src1.bit[9:31]
end
```

Let XT be the sum 32×TX + T.
Let XA be the sum 32×AX + A.
Let XB be the sum 32×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].

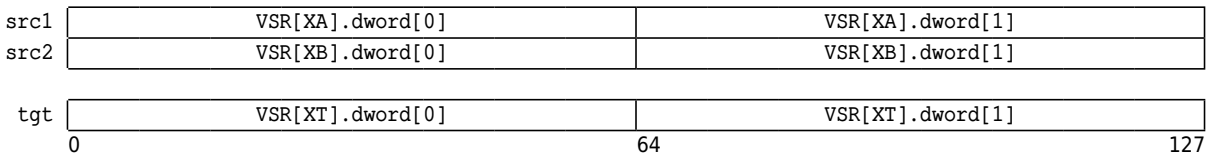
The contents of bits 24:31 of src2 are placed into bits 1:8 of word element i of VSR[XT].

The contents of bits 9:31 of src1 are placed into bits 9:31 of word element i of VSR[XT].

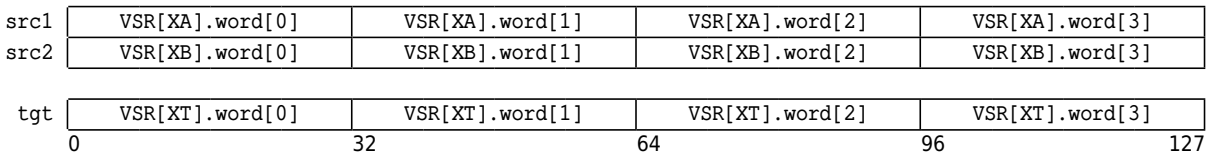
Special Registers Altered:

None

VSR Data Layout for xviexpdp



VSR Data Layout for xviexpsp



VSX Vector Test Data Class Double-Precision XX2-form

xvtstdcdp XT,XB,DCMX

60	T	dx	B	15	dc	5	dm	BX	TX
0	6	11	16	21	25	26	29	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

DCMX ← dc || dm || dx
XT ← 32×TX + T

do i = 0 to 1
   src            ← VSR[32×BX+B].dword[i]
   sign           ← src.bit[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)

   match ←
      (DCMX.bit[0] & class.NaN)              |
      (DCMX.bit[1] & class.Infinity & !sign) |
      (DCMX.bit[2] & class.Infinity &  sign) |
      (DCMX.bit[3] & class.Zero     & !sign) |
      (DCMX.bit[4] & class.Zero     &  sign) |
      (DCMX.bit[5] & class.Denormal & !sign) |
      (DCMX.bit[6] & class.Denormal &  sign)

   if match = 1 then
      VSR[XT].dword[i] ← 0xFFFF_FFFF_FFFF_FFFF
   else
      VSR[XT].dword[i] ← 0x0000_0000_0000_0000
end
```

Let XB be the sum 32×BX + B.
Let XT be the sum 32×TX + T.
Let DCMX be the value dc concatenated with dm concatenated with dx.

For each integer value i from 0 to 1, do the following.

Let src be the double-precision floating-point value in doubleword element i of VSR[XB].

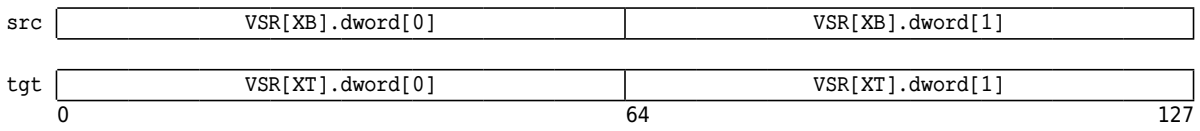
If src matches one of the 7 possible data classes specified by DCMX (Data Class Mask), the contents of doubleword element i of VSR[XT] are set to 0xFFFF_FFFF_FFFF_FFFF. Otherwise, the contents of doubleword element i of VSR[XT] are set to 0x0000_0000_0000_0000.

DCMX bit	Data Class
0	NaN
1	+Infinity
2	−Infinity
3	+Zero
4	−Zero
5	+Denormal
6	−Denormal

Special Registers Altered:

None

VSR Data Layout for xvtstdcdp



VSX Vector Test Data Class Single-Precision XX2-form

xvtstdcsp XT,XB,DCMX

60	T	dx	B	13	dc	5	dm	BX	TX
0	6	11	16	21	25	26	29	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

DCMX ← dc || dm || dx

do i = 0 to 3
   src            ← VSR[32×BX+B].word[i]
   sign           ← src.bit[0]
   exponent       ← src.bit[1:8]
   fraction       ← src.bit[9:31]

   class.Infinity ← (exponent = 0xFF) & (fraction  =
0)
   class.NaN      ← (exponent = 0xFF) & (fraction !=
0)
   class.Zero     ← (exponent = 0x00) & (fraction  =
0)
   class.Denormal ← (exponent = 0x00) & (fraction !=
0)

   match ←
      (DCMX.bit[0] & class.NaN)              |
      (DCMX.bit[1] & class.Infinity & !sign) |
      (DCMX.bit[2] & class.Infinity &  sign) |
      (DCMX.bit[3] & class.Zero     & !sign) |
      (DCMX.bit[4] & class.Zero     &  sign) |
      (DCMX.bit[5] & class.Denormal & !sign) |
      (DCMX.bit[6] & class.Denormal &  sign)

   if match = 1 then
      VSR[32×TX+T].dword[i] ← 0xFFFF_FFFF
   else
      VSR[32×TX+T].dword[i] ← 0x0000_0000
end
```

Let XB be the sum 32×BX + B.
Let XT be the sum 32×TX + T.
Let DCMX be the value dc concatenated with dm concatenated with dx.

For each integer value i from 0 to 3, do the following.

Let src be the single-precision floating-point value in word element i of VSR[XB].

If src matches one of the 7 possible data classes specified by DCMX (Data Class Mask), the contents of word element i of VSR[XT] are set to 0xFFFF_FFFF. Otherwise, the contents of word element i of VSR[XT] are set to 0x0000_0000.

DCMX bit	Data Class
0	NaN
1	+Infinity
2	−Infinity
3	+Zero
4	−Zero
5	+Denormal
6	−Denormal

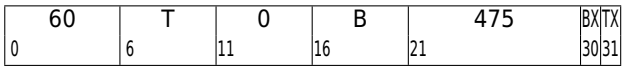
Special Registers Altered:

None

VSR Data Layout for xvtstdcsp

VSX Vector Extract Exponent Double-Precision XX2-form

xvxexpdp XT,XB




```
if MSR.VSX=0 then VSX_Unavailable()

do i = 0 to 1
   src ← VSR[32×BX+B].dword[i]
   VSR[32×TX+T].dword[i] ← EXTZ64(src.bit[1:11])
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].

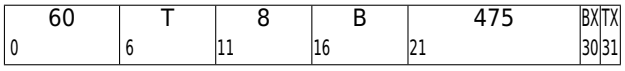
The value of the exponent field in src is placed into doubleword element i of VSR[XT] in unsigned integer format.

Special Registers Altered:

None

VSX Vector Extract Exponent Single-Precision XX2-form

xvxexpsp XT,XB




```
if MSR.VSX=0 then VSX_Unavailable()

do i = 0 to 3
   src ← VSR[32×BX+B].word[i]
   VSR[32×TX+T].word[i] ← EXTZ32(src.bit[1:8])
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.

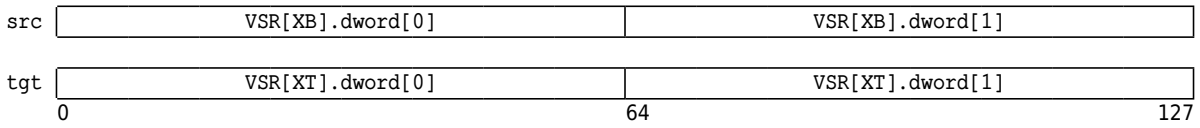
Let src be the single-precision floating-point value in word element i of VSR[XB].

The value of the exponent field in src is placed into word element i of VSR[XT] in unsigned integer format.

Special Registers Altered:

None

VSR Data Layout for xvxexpdp



VSR Data Layout for xvxexpsp

VSX Vector Extract Significand Double-Precision XX2-form

xvxsigdp XT,XB

60	T	1	B	475	BX	TX
0	6	11	16	21	30	31

```
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 | (0x001 || 52)0)

   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

VSX Vector Extract Significand Single-Precision XX2-form

xvxsigsp XT,XB

60	T	9	B	475	BX	TX
0	6	11	16	21	30	31

```
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.

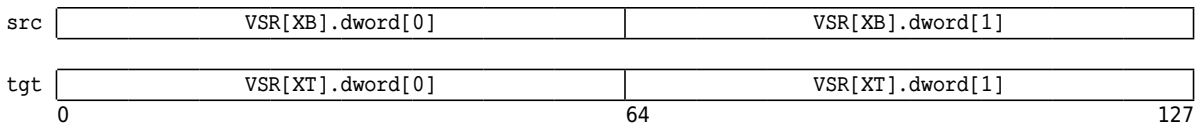
Let src be the single-precision floating-point value in word element i of VSR[XB].

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 xvxsigdp



VSR Data Layout for xvxsigsp

7.6.2.12 VSX Matrix-Multiply Assist (MMA) Instructions

The MMA facility is optional. Software that uses this facility should test for its availability and provide an alternate execution path.

7.6.2.12.1 VSX Accumulator Move Instructions

VSX Move From Accumulator X-form

xxmfacc AS

31	AS	//	0	///	177	/
0	6	9	11	16	21	31

```
if MSR.VSX=0 then VSX_Unavailable()

VSR[4×AS]   ← ACC[AS][0]
VSR[4×AS+1] ← ACC[AS][1]
VSR[4×AS+2] ← ACC[AS][2]
VSR[4×AS+3] ← ACC[AS][3]
```

For each integer value `i` from 0 to 3, the contents of row `i` of `ACC[AS]` are placed into `VSR[4×AS+i]`.

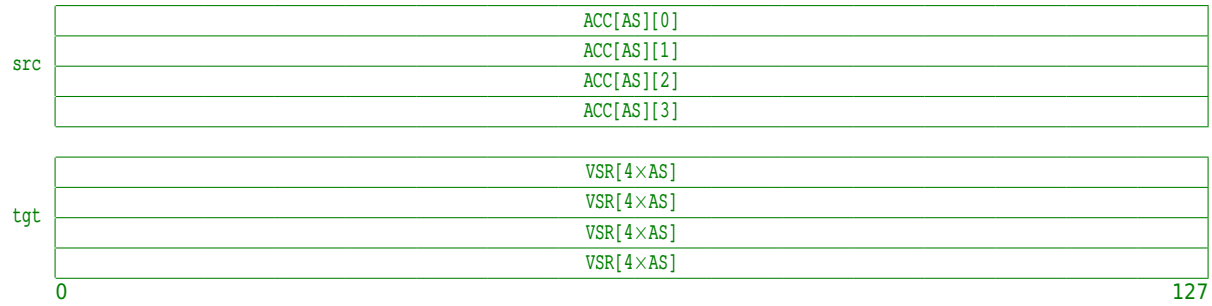
Special Registers Altered:

None

Programming Note

During extended periods of the execution of an application when there isn't any active use of the Accumulators and *VSX Vector GER* instructions, hardware may deactivate these facilities for power savings. Once deactivated, while any attempted execution of any *xxmfacc*, ***xxmtacc***, ***xxsetaccz***, or *VSX Vector GER* instruction will cause these facilities to become reactivated, this reactivation causes significant delay beyond the normal execution of these instructions. This delay can be avoided by periodically issuing an *xxmfacc* with `AS=0` instruction during extended times that the facilities are not being used to keep the facilities activated. Since the contents of `ACC[0]` will be undefined after the first execution, performance on subsequent executions of `xxmfacc 0` can be expected to be degraded compared to performance when the contents of `ACC[0]` are defined. As such, to keep the facilities activated, `xxmfacc 0` should be used with attention to performance implications.

VSR Data Layout for xxmfacc



VSX Move To Accumulator X-form

xxmtacc AT

31	AT	//	1	///	177	/
0	6	9	11	16	21	31

```
if MSR.VSX=0 then VSX_Unavailable()

ACC[AT][0] ← VSR[4×AT]
ACC[AT][1] ← VSR[4×AT+1]
ACC[AT][2] ← VSR[4×AT+2]
ACC[AT][3] ← VSR[4×AT+3]
```

For each integer value `i` from 0 to 3, the contents of `VSR[4×AT+i]` are placed into row `i` of `ACC[AT]`.

Special Registers Altered:

None

VSX Set Accumulator to Zero X-form

xxsetaccz AT

31	AT	//	3	///	177	/
0	6	9	11	16	21	31

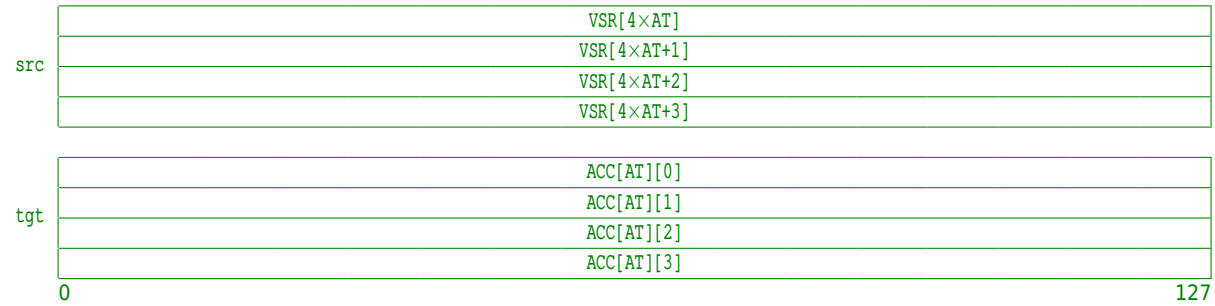
```
if MSR.VSX=0 then VSX_Unavailable()

ACC[AT][0] ← 0x0000_0000_0000_0000_0000_0000_0000_0000
ACC[AT][1] ← 0x0000_0000_0000_0000_0000_0000_0000_0000
ACC[AT][2] ← 0x0000_0000_0000_0000_0000_0000_0000_0000
ACC[AT][3] ← 0x0000_0000_0000_0000_0000_0000_0000_0000
```

For each integer value `i` from 0 to 3, the contents of row `i` of `ACC[AT]` are set to 0.

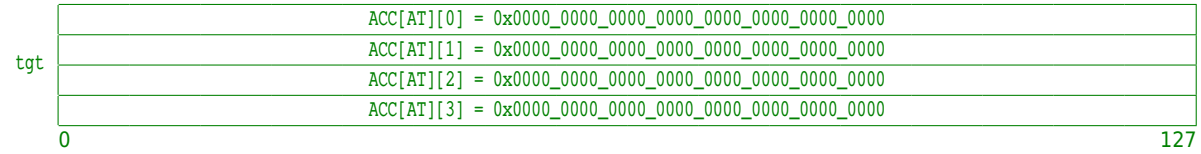
Special Registers Altered:

None

VSR Data Layout for xxmtacc



VSR Data Layout for xxsetaccz



7.6.2.12.2 VSX Binary Integer Outer-Product Instructions

VSX Vector 16-bit Signed Integer GER (rank-2 update) XX3-form

xvi16ger2 AT,XA,XB

59	AT	//	A	B	75	AX	BX	/
0	6	9	11	16	21	29	30	31

VSX Vector 16-bit Signed Integer GER (rank-2 update) Positive multiply, Positive accumulate XX3-form

xvi16ger2pp AT,XA,XB

59	AT	//	A	B	107	AX	BX	/
0	6	9	11	16	21	29	30	31

Prefixed Masked VSX Vector 16-bit Signed Integer GER (rank-2 update) MMIRR:XX3-form

pmxvi16ger2 AT,XA,XB,XMSK,YMSK,PMSK

Prefix:

1	3	9	//	/	/	PMSK	///	XMSK	YMSK
0	6	8	12	14	15	16	18	24	28 31

Suffix:

59	AT	//	A	B	75	AX	BX	/
0	6	9	11	16	21	29	30	31

Prefixed Masked VSX Vector 16-bit Signed Integer GER (rank-2 update) Positive multiply, Positive accumulate MMIRR:XX3-form

pmxvi16ger2pp AT,XA,XB,XMSK,YMSK,PMSK

Prefix:

1	3	9	//	/	/	PMSK	///	XMSK	YMSK
0	6	8	12	14	15	16	18	24	28 31

Suffix:

59	AT	//	A	B	107	AX	BX	/
0	6	9	11	16	21	29	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

if ''xvi16ger2s'' | ''xvi16ger2spp'' then do
   PMSK ← 0b11
   XMSK ← 0b1111
   YMSK ← 0b1111
end

do i = 0 to 3
   do j = 0 to 3
      if XMSK.bit[i] & YMSK.bit[j] then do
         prod0 ← (PMSK.bit[0]=0) ? 0 : EXTS(VSR[32×AX+A].word[i].hword[0]) *
                                       EXTS(VSR[32×BX+B].word[j].hword[0])
         prod1 ← (PMSK.bit[1]=0) ? 0 : EXTS(VSR[32×AX+A].word[i].hword[1]) *
                                       EXTS(VSR[32×BX+B].word[j].hword[1])

         psum  ← prod0 + prod1

         if ''[pm]xvi16ger2''   then ACC[AT][i].word[j] ← CHOP32(psum)
         if ''[pm]xvi16ger2pp'' then ACC[AT][i].word[j] ←
                                       CHOP32(psum + EXTS(ACC[AT][i].word[j])
      end
      else
         ACC[AT][i][j] ← 0x0000_0000
   end
end
```

Let XA be the value of 32×AX + A. If XA is in the range 4×AT to 4×AT+3, the instruction form is invalid.
Let XB be the value of 32×BX + B. If XB is in the range 4×AT to 4×AT+3, the instruction form is invalid.

Let the contents of ACC[AT] be a 4×4 matrix of 32-bit signed integer values.

For ***xvi16ger2*** or ***xvi16ger2pp***, let PMSK=0b11, XMSK=0b1111, and YMSK=0b1111.

For each integer value i from 0 to 3 and each integer value j from 0 to 3, do the following.

If bit i of XMSK is equal to 1 and bit j of YMSK is equal to 1, do the following.

If bit 0 of PMSK is equal to 1, let prod0 be the product of the 16-bit signed integer value in halfword 0 of word element i of VSR[XA] and the 16-bit signed integer value in halfword 0 of word element j of VSR[XB]. Otherwise, let prod0 be the value 0.

If bit 1 of PMSK is equal to 1, let prod1 be the product of the 16-bit signed integer value in halfword 1 of word element i of VSR[XA] and the 16-bit signed integer value in halfword 1 of word element j of VSR[XB]. Otherwise, let prod1 be the value 0.

Let psum be the sum of prod0 and prod1.

For [***pm***]***xvi16ger2***, psum is placed into word element j of ACC[AT][i] in 32-bit signed integer format.

For [***pm***]***xvi16ger2pp***, psum is added to the 32-bit signed integer value in word element j of ACC[AT][i], and the result is placed into word element j of ACC[AT][i] in 32-bit signed integer format.

Otherwise, ACC[AT][i][j] is set to 0x0000_0000.

Special Registers Altered:

None

Register Operand Data Layout for [pm]xvi16ger2[pp]

VSR[XA]	X[0][0]	X[0][1]	X[1][0]	X[1][1]	X[2][0]	X[2][1]	X[3][0]	X[3][1]
VSR[XB]	Y[0][0]	Y[1][0]	Y[0][1]	Y[1][1]	Y[0][2]	Y[1][2]	Y[0][3]	Y[1][3]

ACC[AT][0]	T[0][0]		T[0][1]		T[0][2]		T[0][3]	
ACC[AT][1]	T[1][0]		T[1][1]		T[1][2]		T[1][3]	
ACC[AT][2]	T[2][0]		T[2][1]		T[2][2]		T[2][3]	
ACC[AT][3]	T[3][0]		T[3][1]		T[3][2]		T[3][3]	
	0	16	32	48	64	80	96	112 127

Programming Note

Let X be the 4×2 matrix of 16-bit signed integer values contained in VSR[XA] in row-major format.
Let Y be the 4×2 matrix of 16-bit signed integer values contained in VSR[XB] in row-major format.
Let ACC[AT] be the Accumulator containing a 4×4 matrix of 32-bit signed integer values.

[***pm***]***xvi16ger2*** performs the following form of accumulation of two outer products (rank 2 update).

```
ACC[i][j] = si32_CHOP( EXTS(X[i][0]) * EXTS(Y[j][0]) +
                       EXTS(X[i][1]) * EXTS(Y[j][1]) )
```

[***pm***]***xvi16ger2pp*** performs the following form of accumulation of two outer products (rank 2 update).

```
ACC[i][j] = si32_CHOP( EXTS(X[i][0]) * EXTS(Y[j][0]) +
                       EXTS(X[i][1]) * EXTS(Y[j][1]) +
                       EXTS(ACC[i][J]) )
```

VSX Vector 16-bit Signed Integer GER (rank-2 update) with Saturation XX3-form

xvi16ger2s AT,XA,XB

59	AT	//	A	B	43	AX	BX	/
0	6	9	11	16	21	29	30	31

Prefixed Masked VSX Vector 16-bit Signed Integer GER (rank-2 update) with Saturation MMIRR:XX3-form

pmxvi16ger2s AT,XA,XB,XMSK,YMSK,PMSK

Prefix:

1	3	9	//	/	/	PMSK	///	XMSK	YMSK
0	6	8	12	14	15	16	18	24	28 31

Suffix:

59	AT	//	A	B	43	AX	BX	/
0	6	9	11	16	21	29	30	31

VSX Vector 16-bit Signed Integer GER (rank-2 update) with Saturation Positive multiply, Positive accumulate XX3-form

xvi16ger2spp AT,XA,XB

59	AT	//	A	B	42	AX	BX	/
0	6	9	11	16	21	29	30	31

Prefixed Masked VSX Vector 16-bit Signed Integer GER (rank-2 update) with Saturation Positive multiply, Positive accumulate MMIRR:XX3-form

pmxvi16ger2spp AT,XA,XB,XMSK,YMSK,PMSK

Prefix:

1	3	9	//	/	/	PMSK	///	XMSK	YMSK
0	6	8	12	14	15	16	18	24	28 31

Suffix:

59	AT	//	A	B	42	AX	BX	/
0	6	9	11	16	21	29	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

if ''xvi16ger2s'' | ''xvi16ger2spp'' then do
   PMSK ← 0b11
   XMSK ← 0b1111
   YMSK ← 0b1111
end

sat_flag ← 0

do i = 0 to 3
   do j = 0 to 3
      if XMSK.bit[i] & YMSK.bit[j] then do
         prod0 ← (PMSK.bit[0]=0) ? 0 : EXTS(VSR[32×AX+A].word[i].hword[0]) *
                                       EXTS(VSR[32×BX+B].word[j].hword[0])
         prod1 ← (PMSK.bit[1]=0) ? 0 : EXTS(VSR[32×AX+A].word[i].hword[1]) *
                                       EXTS(VSR[32×BX+B].word[j].hword[1])

         psum  ← prod0 + prod1

         if ''[pm]xvi16ger2s'' then
            ACC[AT][i].word[j] ← si32_CLAMP( psum )
         if ''[pm]xvi16ger2spp'' then
            ACC[AT][i].word[j] ← si32_CLAMP( psum + EXTS(ACC[AT][i].word[j] )

         if sat_flag=1 then VSCR.SAT ← 1
      end
      else
         ACC[AT][i][j] ← 0x0000_0000
   end
end
```

Let XA be the value of 32×AX + A. If XA is in the range 4×AT to 4×AT+3, the instruction form is invalid.
Let XB be the value of 32×BX + B. If XB is in the range 4×AT to 4×AT+3, the instruction form is invalid.

Let the contents of ACC[AT] be a 4×4 matrix of 32-bit signed integer values.

For ***xvi16ger2s*** or ***xvi16ger2spp***, let PMSK=0b11, XMSK=0b1111, and YMSK=0b1111.

For each integer value i from 0 to 3 and each integer value j from 0 to 3, do the following.

If bit i of XMSK is equal to 1 and bit j of YMSK is equal to 1, do the following.

If bit 0 of PMSK is equal to 1, let prod0 be the product of the 16-bit signed integer value in halfword 0 of word element i of VSR[XA] and the 16-bit signed integer value in halfword 0 of word element j of VSR[XB]. Otherwise, let prod0 be the value 0.

If bit 1 of PMSK is equal to 1, let prod1 be the product of the 16-bit signed integer value in halfword 1 of word element i of VSR[XA] and the 16-bit signed integer value in halfword 1 of word element j of VSR[XB]. Otherwise, let prod1 be the value 0.

Let sum be the sum of prod0 and prod1.

For [***pm***]***xvi16ger2s***, let result be psum.

For [***pm***]***xvi16ger2spp***, let result be the sum of psum to the 32-bit signed integer value in word element j of ACC[AT][i].

If result is less than -2^{31}, result saturates to -2^{31} and SAT is set to 1.

If result is greater than $2^{31} - 1$, result saturates to $2^{31} - 1$ and SAT is set to 1.

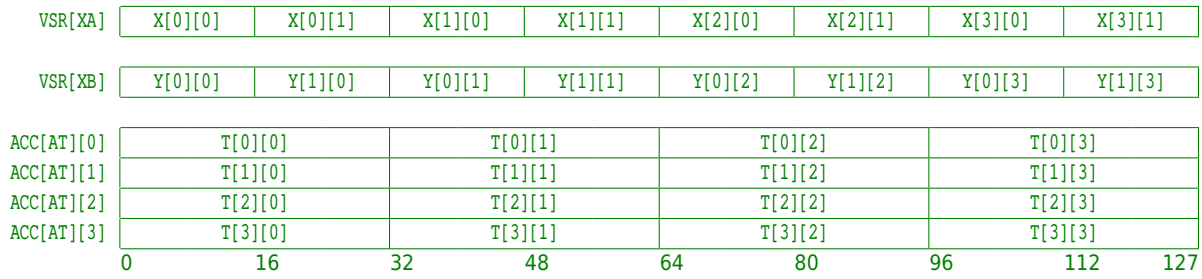
result is placed into word element j of ACC[AT][i] in 32-bit signed integer format.

Otherwise, ACC[AT][i][j] is set to 0x0000_0000.

Special Registers Altered:

Register	*Field(s)*
VSCR	SAT

Register Operand Data Layout for [pm]xvi16ger2s[pp]



Programming Note

Let X be the 4×2 matrix of 16-bit signed integer values contained in VSR[XA] in row-major format.
Let Y be the 4×2 matrix of 16-bit signed integer values contained in VSR[XB] in row-major format.
Let ACC[AT] be the Accumulator containing a 4×4 matrix of 32-bit signed-integer values.

[***pm***]***xvi16ger2s*** performs the following form of accumulation of two outer products (rank 2 update).

```
ACC[AT][i][j] = si32_CLAMP( EXTS(X[i][0]) * EXTS(Y[j][0]) +
                            EXTS(X[i][1]) * EXTS(Y[j][1]) )
```

[***pm***]***xvi16ger2spp*** performs the following form of accumulation of two outer products (rank 2 update).

```
ACC[AT][i][j] = si32_CLAMP( EXTS(X[i][0]) * EXTS(Y[j][0]) +
                            EXTS(X[i][1]) * EXTS(Y[j][1]) +
                            EXTS(ACC[AT][i][1]) )
```

VSX Vector 4-bit Signed Integer GER (rank-8 update) XX3-form

xvi4ger8 AT,XA,XB

59	AT	//	A	B	35	AX	BX	/
0	6	9	11	16	21	29	30	31

Prefixed Masked VSX Vector 4-bit Signed Integer GER (rank-8 update) MMIRR:XX3-form

pmxvi4ger8 AT,XA,XB,XMSK,YMSK,PMSK

Prefix:

1	3	9	//	/	/	PMSK	XMSK	YMSK
0	6	8	12	14	15	16	24	28 31

Suffix:

59	AT	//	A	B	35	AX	BX	/
0	6	9	11	16	21	29	30	31

VSX Vector 4-bit Signed Integer GER (rank-8 update) Positive multiply, Positive accumulate XX3-form

xvi4ger8pp AT,XA,XB

59	AT	//	A	B	34	AX	BX	/
0	6	9	11	16	21	29	30	31

Prefixed Masked VSX Vector 4-bit Signed Integer GER (rank-8 update) Positive multiply, Positive accumulate MMIRR:XX3-form

pmxvi4ger8pp AT,XA,XB,XMSK,YMSK,PMSK

Prefix:

1	3	9	//	/	/	PMSK	XMSK	YMSK
0	6	8	12	14	15	16	24	28 31

Suffix:

59	AT	//	A	B	34	AX	BX	/
0	6	9	11	16	21	29	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

if ''xvi4ger8'' | ''xvi4ger8pp'' then do
   PMSK ← 0b11111111
   XMSK ← 0b1111
   YMSK ← 0b1111
end

do i = 0 to 3
   do j = 0 to 3
      if XMSK.bit[i] & YMSK.bit[j] then do
         prod0 ← (PMSK.bit[0]=0) ? 0 : EXTS(VSR[32×AX+A].word[i].nibble[0]) *
                                       EXTS(VSR[32×BX+B].word[j].nibble[0])
         prod1 ← (PMSK.bit[1]=0) ? 0 : EXTS(VSR[32×AX+A].word[i].nibble[1]) *
                                       EXTS(VSR[32×BX+B].word[j].nibble[1])
         prod2 ← (PMSK.bit[2]=0) ? 0 : EXTS(VSR[32×AX+A].word[i].nibble[2]) *
                                       EXTS(VSR[32×BX+B].word[j].nibble[2])
         prod3 ← (PMSK.bit[3]=0) ? 0 : EXTS(VSR[32×AX+A].word[i].nibble[3]) *
                                       EXTS(VSR[32×BX+B].word[j].nibble[3])
         prod4 ← (PMSK.bit[4]=0) ? 0 : EXTS(VSR[32×AX+A].word[i].nibble[4]) *
                                       EXTS(VSR[32×BX+B].word[j].nibble[4])
         prod5 ← (PMSK.bit[5]=0) ? 0 : EXTS(VSR[32×AX+A].word[i].nibble[5]) *
                                       EXTS(VSR[32×BX+B].word[j].nibble[5])
         prod6 ← (PMSK.bit[6]=0) ? 0 : EXTS(VSR[32×AX+A].word[i].nibble[6]) *
                                       EXTS(VSR[32×BX+B].word[j].nibble[6])
         prod7 ← (PMSK.bit[7]=0) ? 0 : EXTS(VSR[32×AX+A].word[i].nibble[7]) *
                                       EXTS(VSR[32×BX+B].word[j].nibble[7])

         psum  ← prod0 + prod1 + prod2 + prod3 + prod4 + prod5 + prod6 + prod7

         if ''[pm]xvi4ger8''   then ACC[AT][i].word[j] ← CHOP32( psum )
         if ''[pm]xvi4ger8pp'' then ACC[AT][i].word[j] ←
                                    CHOP32( psum + EXTS(ACC[AT][i].word[j]) )
      end
      else
         ACC[AT][i].word[j] ← 0x0000_0000
   end
end
```

Let XA be the value of 32×AX + A. If XA is in the range 4×AT to 4×AT+3, the instruction form is invalid.
Let XB be the value of 32×BX + B. If XB is in the range 4×AT to 4×AT+3, the instruction form is invalid.

Let the contents of ACC[AT] be a 4×4 matrix of 32-bit signed integer values.

For ***xvi4ger8*** or ***xvi4ger8pp***, let PMSK=0b11111111, XMSK=0b1111, and YMSK=0b1111.

For each integer value i from 0 to 3, and each integer value j from 0 to 3, do the following.

If bit i of XMSK is equal to 1 and bit j of YMSK is equal to 1, do the following.

If bit 0 of PMSK is equal to 1, let prod0 be the product of the 4-bit signed integer value in nibble 0 of word element i of VSR[XA] (X[i][0]) and the 4-bit signed integer value in byte 0 of word element j of VSR[XB] (Y[j][0]), sign-extended to 32 bits. Otherwise, let prod0 be the value 0.

If bit 1 of PMSK is equal to 1, let prod1 be the product of the 4-bit signed integer value in nibble 1 of word element i of VSR[XA] (X[i][1]) and the 4-bit signed integer value in nibble 1 of word element j of VSR[XB] (Y[j][1]), sign-extended to 32 bits. Otherwise, let prod1 be the value 0.

If bit 2 of PMSK is equal to 1, let prod2 be the product of the 4-bit signed integer value in nibble 2 of word element i of VSR[XA] (X[i][2]) and the 4-bit signed integer value in nibble 2 of word element j of VSR[XB] (Y[j][2]), sign-extended to 32 bits. Otherwise, let prod2 be the value 0.

If bit 3 of PMSK is equal to 1, let prod3 be the product of the 4-bit signed integer value in nibble 3 of word element i of VSR[XA] (X[i][3]) and the 4-bit signed integer value in nibble 3 of word element j of VSR[XB] (Y[j][3]), sign-extended to 32 bits. Otherwise, let prod3 be the value 0.

If bit 4 of PMSK is equal to 1, let prod4 be the product of the 4-bit signed integer value in nibble 4 of word element i of VSR[XA] (X[i][4]) and the 4-bit signed integer value in nibble 4 of word element j of VSR[XB] (Y[j][4]), sign-extended to 32 bits. Otherwise, let prod4 be the value 0.

If if bit 5 of PMSK is equal to 1, let prod5 be the product of the 4-bit signed integer value in nibble 5 of word element i of VSR[XA] (X[i][5]) and the 4-bit signed integer value in nibble 5 of word element j of VSR[XB] (Y[j][5]), sign-extended to 32 bits. Otherwise, let prod5 be the value 0.

If bit 6 of PMSK is equal to 1, let prod6 be the product of the 4-bit signed integer value in nibble 6 of word element i of VSR[XA] (X[i][6]) and the 4-bit signed integer value in nibble 6 of word element j of VSR[XB] (Y[j][6]), sign-extended to 32 bits. Otherwise, let prod6 be the value 0.

If bit 7 of PMSK is equal to 1, let prod7 be the product of the 4-bit signed integer value in nibble 7 of word element i of VSR[XA] (X[i][7]) and the 4-bit signed integer value in nibble 7 of word element j of VSR[XB] (Y[j][7]), sign-extended to 32 bits. Otherwise, let prod7 be the value 0.

Let psum be the sum of prod0, prod1, prod2, prod3, prod4, prod5, prod6, and prod7.

For [***pm***]***xvi4ger8***, psum is placed into word element j of ACC[AT][i] in 32-bit signed integer format.

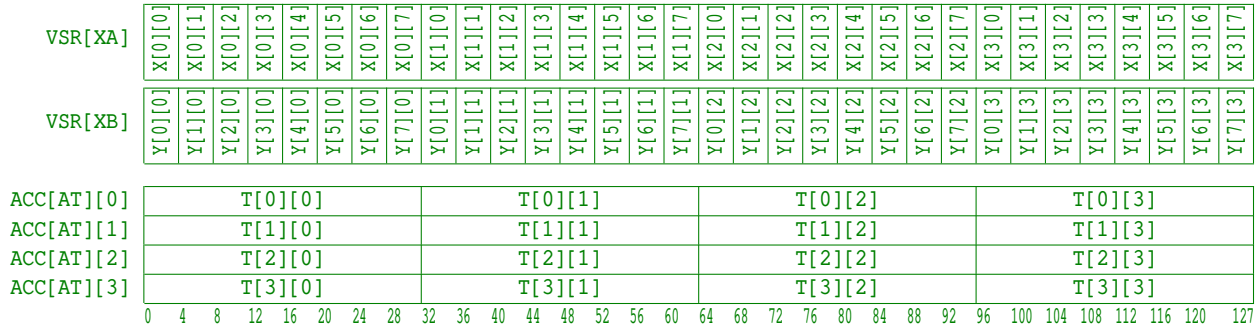
For [***pm***]***xvi4ger8pp***, psum is added to the 32-bit signed integer value in word element j of ACC[AT][i], and the result is placed into word element j of ACC[AT][i] in 32-bit signed integer format.

Otherwise, ACC[AT][i][j] is set to 0x0000_0000.

Special Registers Altered:

None

Register Operand Data Layout for [pm]xvi4ger8[pp]



Programming Note

Let X be the 8×4 matrix of 4-bit signed integer values contained in VSR[XA] in row-major format.
Let Y be the 8×4 matrix of 4-bit signed integer values contained in VSR[XB] in row-major format.
Let ACC[AT] be the Accumulator containing a 4×4 matrix of 32-bit signed-integer values.

[***pm***]***xvi4ger8*** performs the following form of accumulation of eight outer products (rank 8 update).

```
ACC[AT][i][j] = si32_CHOP( EXTS(X[i][0]) * EXTS(Y[j][0]) +
                           EXTS(X[i][1]) * EXTS(Y[j][1]) +
                           EXTS(X[i][2]) * EXTS(Y[j][2]) +
                           EXTS(X[i][3]) * EXTS(Y[j][3]) +
                           EXTS(X[i][4]) * EXTS(Y[j][4]) +
                           EXTS(X[i][5]) * EXTS(Y[j][5]) +
                           EXTS(X[i][6]) * EXTS(Y[j][6]) +
                           EXTS(X[i][7]) * EXTS(Y[j][7]) )
```

[***pm***]***xvi4ger8pp*** performs the following form of accumulation of eight outer products (rank 8 update).

```
ACC[AT][i][j] = si32_CHOP( EXTS(X[i][0]) * EXTS(Y[j][0]) +
                           EXTS(X[i][1]) * EXTS(Y[j][1]) +
                           EXTS(X[i][2]) * EXTS(Y[j][2]) +
                           EXTS(X[i][3]) * EXTS(Y[j][3]) +
                           EXTS(X[i][4]) * EXTS(Y[j][4]) +
                           EXTS(X[i][5]) * EXTS(Y[j][5]) +
                           EXTS(X[i][6]) * EXTS(Y[j][6]) +
                           EXTS(X[i][7]) * EXTS(Y[j][7]) +
                           EXTS(ACC[AT][i][j]) )
```

VSX Vector 8-bit Signed/Unsigned Integer GER (rank-4 update) XX3-form

xvi8ger4 AT,XA,XB

59	AT	//	A	B	3	AX	BX	/
0	6	9	11	16	21	29	30	31

Prefixed Masked VSX Vector 8-bit Signed/Unsigned Integer GER (rank-4 update) MMIRR:XX3-form

pmxvi8ger4 AT,XA,XB,XMSK,YMSK,PMSK

Prefix:

1	3	9	//	/	/	PMSK	///	XMSK	YMSK
0	6	8	12	14	15	16	20	24	28 31

Suffix:

59	AT	//	A	B	3	AX	BX	/
0	6	9	11	16	21	29	30	31

VSX Vector 8-bit Signed/Unsigned Integer GER (rank-4 update) Positive multiply, Positive accumulate XX3-form

xvi8ger4pp AT,XA,XB

59	AT	//	A	B	2	AX	BX	/
0	6	9	11	16	21	29	30	31

Prefixed Masked VSX Vector 8-bit Signed/Unsigned Integer GER (rank-4 update) Positive multiply, Positive accumulate MMIRR:XX3-form

pmxvi8ger4pp AT,XA,XB,XMSK,YMSK,PMSK

Prefix:

1	3	9	//	/	/	PMSK	///	XMSK	YMSK
0	6	8	12	14	15	16	20	24	28 31

Suffix:

59	AT	//	A	B	2	AX	BX	/
0	6	9	11	16	21	29	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

if ''xvi8ger4'' | ''xvi8ger4pp'' then do
   PMSK ← 0b1111
   XMSK ← 0b1111
   YMSK ← 0b1111
end

do i = 0 to 3
   do j = 0 to 3
      if XMSK.bit[i] & YMSK.bit[j] then do
         prod0 ← (PMSK.bit[0]=0) ? 0 : EXTS(VSR[32×AX+A].word[i].byte[0]) *
                                       EXTZ(VSR[32×BX+B].word[j].byte[0])
         prod1 ← (PMSK.bit[1]=0) ? 0 : EXTS(VSR[32×AX+A].word[i].byte[1]) *
                                       EXTZ(VSR[32×BX+B].word[j].byte[1])
         prod2 ← (PMSK.bit[2]=0) ? 0 : EXTS(VSR[32×AX+A].word[i].byte[2]) *
                                       EXTZ(VSR[32×BX+B].word[j].byte[2])
         prod3 ← (PMSK.bit[3]=0) ? 0 : EXTS(VSR[32×AX+A].word[i].byte[3]) *
                                       EXTZ(VSR[32×BX+B].word[j].byte[3])

         psum  ← prod0 + prod1 + prod2 + prod3

         if ''[pm]xvi8ger4''   then ACC[AT][i].word[j] ← CHOP32( psum )
         if ''[pm]xvi8ger4pp'' then ACC[AT][i].word[j] ←
                                       CHOP32( psum + EXTS(ACC[AT][i].word[j]) )
      end
      else
         ACC[AT][i][j] ← 0x0000_0000
   end
end
```

Let XA be the value of 32×AX + A. If XA is in the range 4×AT to 4×AT+3, the instruction form is invalid.
Let XB be the value of 32×BX + B. If XB is in the range 4×AT to 4×AT+3, the instruction form is invalid.

Let the contents of ACC[AT] be a 4×4 matrix of 32-bit signed integer values.

For ***xvi8ger4*** or ***xvi8ger4pp***, let PMSK=0b1111, XMSK=0b1111, and YMSK=0b1111.

For each integer value i from 0 to 3, and each integer value j from 0 to 3, do the following.

If bit i of XMSK is equal to 1 and bit j of YMSK is equal to 1, do the following.

If bit 0 of PMSK is equal to 1, let prod0 be the product of the 8-bit signed integer value in byte 0 of word element i of VSR[XA] (X[i][0]) and the 8-bit unsigned integer value in byte 0 of word element j of VSR[XB] (Y[j][0]), sign-extended to 32 bits. Otherwise, let prod0 be the value 0.

If bit 1 of PMSK is equal to 1, let prod1 be the product of the 8-bit signed integer value in byte 1 of word element i of VSR[XA] (X[i][1]) and the 8-bit unsigned integer value in byte 1 of word element j of VSR[XB] (Y[j][1]), sign-extended to 32 bits. Otherwise, let prod1 be the value 0.

If bit 2 of PMSK is equal to 1, let prod2 be the product of the 8-bit signed integer value in byte 2 of word element i of VSR[XA] (X[i][2]) and the 8-bit unsigned integer value in byte 2 of word element j of VSR[XB] (Y[j][2]), sign-extended to 32 bits. Otherwise, let prod2 be the value 0.

If bit 3 of PMSK is equal to 1, let prod3 be the product of the 8-bit signed integer value in byte 3 of word element i of VSR[XA] (X[i][3]) and the 8-bit unsigned integer value in byte 3 of word element j of VSR[XB] (Y[j][3]), sign-extended to 32 bits. Otherwise, let prod3 be the value 0.

Let sum be the sum of prod0, prod1, prod2, and prod3.

For [***pm***]***xvi8ger4***, psum is placed into word element j of ACC[AT][i] in 32-bit signed integer format.

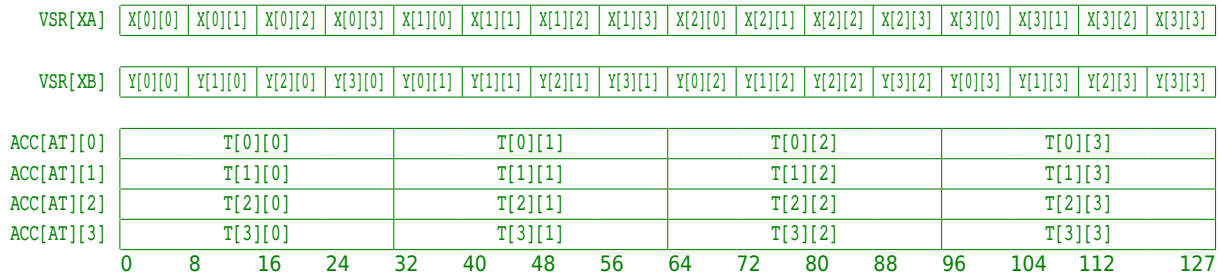
For [***pm***]***xvi8ger4pp***, psum is added to the 32-bit signed integer value in word element j of ACC[AT][i], and the result is placed into word element j of ACC[AT][i] in 32-bit signed integer format.

Otherwise, word element j of ACC[AT][i] is set to 0x0000_0000.

Special Registers Altered:

None

Register Operand Data Layout for [pm]xvi8ger4[pp]



Programming Note

Let X be the 4×4 matrix of 8-bit signed integer values contained in VSR[XA] in row-major format.
Let Y be the 4×4 matrix of 8-bit unsigned integer values contained in VSR[XB] in row-major format.
Let ACC[AT] be the Accumulator containing a 4×4 matrix of 32-bit signed-integer values.

[***pm***]***xvi8ger4*** performs the following form of accumulation of four outer products (rank 4 update).

```
ACC[AT][i][j] = si32_CHOP( EXTS(X[i][0]) * EXTZ(Y[j][0]) +
                           EXTS(X[i][1]) * EXTZ(Y[j][1]) +
                           EXTS(X[i][2]) * EXTZ(Y[j][2]) +
                           EXTS(X[i][3]) * EXTZ(Y[j][3]) )
```

[***pm***]***xvi8ger4pp*** performs the following form of accumulation of four outer products (rank 4 update).

```
ACC[AT][i][j] = si32_CHOP( EXTS(X[i][0]) * EXTZ(Y[j][0]) +
                           EXTS(X[i][1]) * EXTZ(Y[j][1]) +
                           EXTS(X[i][2]) * EXTZ(Y[j][2]) +
                           EXTS(X[i][3]) * EXTZ(Y[j][3]) +
                           EXTS(ACC[AT][i][j]) )
```

VSX Vector 8-bit Signed/Unsigned Integer GER (rank-4 update) with Saturation Positive multiply, Positive accumulate XX3-form

xvi8ger4spp AT,XA,XB

59	AT	//	A	B	99	AX	BX	/
0	6	9	11	16	21	29	30	31

Prefixed Masked VSX Vector 8-bit Signed/Unsigned Integer GER (rank-4 update) with Saturation Positive multiply, Positive accumulate MMIRR:XX3-form

pmxvi8ger4spp AT,XA,XB,XMSK,YMSK,PMSK

Prefix:

1	3	9	//	/	/	PMSK	///	XMSK	YMSK
0	6	8	12	14	15	16	20	24	28 31

Suffix:

59	AT	//	A	B	99	AX	BX	/
0	6	9	11	16	21	29	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

if ''xvi8ger4pps'' then do
   PMSK ← 0b1111
   XMSK ← 0b1111
   YMSK ← 0b1111
end

do i = 0 to 3
   do j = 0 to 3
      if XMSK.bit[i] & YMSK.bit[j] then do
         prod0 ← (PMSK.bit[0]=0) ? 0 : EXTS(VSR[32×AX+A].word[i].byte[0]) *
                                       EXTZ(VSR[32×BX+B].word[j].byte[0])
         prod1 ← (PMSK.bit[1]=0) ? 0 : EXTS(VSR[32×AX+A].word[i].byte[1]) *
                                       EXTZ(VSR[32×BX+B].word[j].byte[1])
         prod2 ← (PMSK.bit[2]=0) ? 0 : EXTS(VSR[32×AX+A].word[i].byte[2]) *
                                       EXTZ(VSR[32×BX+B].word[j].byte[2])
         prod3 ← (PMSK.bit[3]=0) ? 0 : EXTS(VSR[32×AX+A].word[i].byte[3]) *
                                       EXTZ(VSR[32×BX+B].word[j].byte[3])

         psum  ← prod0 + prod1 + prod2 + prod3

         ACC[AT][i].word[j] ← si32_CLAMP( psum + EXTS(ACC[AT][i].word[j] )

         if sat_flag=1 then VSCR.SAT ← 1
      end
      else
         ACC[AT][i][j] ← 0x0000_0000
   end
end
```

Let XA be the value of 32×AX + A. If XA is in the range 4×AT to 4×AT+3, the instruction form is invalid.
Let XB be the value of 32×BX + B. If XB is in the range 4×AT to 4×AT+3, the instruction form is invalid.

Let the contents of ACC[AT] be a 4×4 matrix of 32-bit signed integer values.

For ***xvi8ger4spp***, let PMSK=0b1111, XMSK=0b1111, and YMSK=0b1111.

For each integer value i from 0 to 3, and each integer value j from 0 to 3, do the following.

If bit i of XMSK is equal to 1 and bit j of YMSK is equal to 1, do the following.

If bit 0 of PMSK is equal to 1, let prod0 be the product of the 8-bit signed integer value in byte 0 of word element i of VSR[XA] (X[i][0]) and the 8-bit unsigned integer value in byte 0 of word element j of VSR[XB] (Y[j][0]), sign-extended to 32 bits. Otherwise, let prod0 be the value 0.

If bit 1 of PMSK is equal to 1, let prod1 be the product of the 8-bit signed integer value in byte 1 of word element i of VSR[XA] (X[i][1]) and the 8-bit unsigned integer value in byte 1 of word element j of VSR[XB] (Y[j][1]), sign-extended to 32 bits. Otherwise, let prod1 be the value 0.

If bit 2 of PMSK is equal to 1, let prod2 be the product of the 8-bit signed integer value in byte 2 of word element i of VSR[XA] (X[i][2]) and the 8-bit unsigned integer value in byte 2 of word element j of VSR[XB] (Y[j][2]), sign-extended to 32 bits. Otherwise, let prod2 be the value 0.

If bit 3 of PMSK is equal to 1, let prod3 be the product of the 8-bit signed integer value in byte 3 of word element i of VSR[XA] (X[i][3]) and the 8-bit unsigned integer value in byte 3 of word element j of VSR[XB] (Y[j][3]), sign-extended to 32 bits. Otherwise, let prod3 be the value 0.

Let psum be the sum of prod0, prod1, prod2, and prod3.

psum is added to the 32-bit signed integer value in word element j of ACC[AT][i] and the result is placed into word element j of ACC[AT][i] in 32-bit signed integer format.

If the result is less than -2^{31}, the result saturates to -2^{31} and SAT is set to 1.

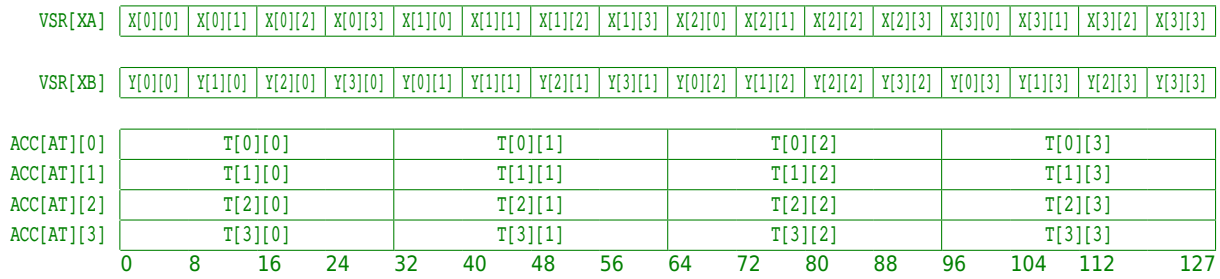
If the result is greater than $2^{31} - 1$, the result saturates to $2^{31} - 1$ and SAT is set to 1.

Otherwise, word element j of ACC[AT][i] is set to 0x0000_0000.

Special Registers Altered:

Register	*Field(s)*
VSCR	SAT

Register Operand Data Layout for [pm]xvi8ger4[pp]



Programming Note

Let X be the 4×4 matrix of 8-bit signed integer values contained in VSR[XA] in row-major format.
Let Y be the 4×4 matrix of 8-bit unsigned integer values contained in VSR[XB] in row-major format.
Let ACC[AT] be the Accumulator containing a 4×4 matrix of 32-bit signed integer values.

[***pm***]***xvi8ger4*** performs the following form of accumulation of four outer products (rank 4 update).

```
ACC[i][j] = si32_CHOP( EXTS(X[i][0]) * EXTZ(Y[j][0]) +
                       EXTS(X[i][1]) * EXTZ(Y[j][1]) +
                       EXTS(X[i][2]) * EXTZ(Y[j][2]) +
                       EXTS(X[i][3]) * EXTZ(Y[j][3]) )
```

Note that a saturating form of the above accumulation is not needed nor provided since the sum of four 16-bit signed integer products cannot overflow the 32-bit signed integer format.

[***pm***]***xvi8ger4spp*** performs the following form of accumulation of four outer products (rank 4 update).

```
ACC[i][j] = si32_CLAMP( EXTS(X[i][0]) * EXTZ(Y[j][0]) +
                        EXTS(X[i][1]) * EXTZ(Y[j][1]) +
                        EXTS(X[i][2]) * EXTZ(Y[j][2]) +
                        EXTS(X[i][3]) * EXTZ(Y[j][3]) +
                        EXTS(ACC[i][j]) )
```

7.6.2.12.3 VSX Binary Floating-Point Outer-Product Instructions

VSX Vector bfloat16 GER (rank-2 update) XX3-form

xvbf16ger2 AT,XA,XB

59	AT	//	A	B	51	AX	BX	/
0	6	9	11	16	21	29	30	31

VSX Vector bfloat16 GER (rank-2 update) Positive multiply, Positive accumulate XX3-form

xvbf16ger2pp AT,XA,XB

59	AT	//	A	B	50	AX	BX	/
0	6	9	11	16	21	29	30	31

VSX Vector bfloat16 GER (rank-2 update) Positive multiply, Negative accumulate XX3-form

xvbf16ger2pn AT,XA,XB

59	AT	//	A	B	178	AX	BX	/
0	6	9	11	16	21	29	30	31

VSX Vector bfloat16 GER (rank-2 update) Negative multiply, Positive accumulate XX3-form

xvbf16ger2np AT,XA,XB

59	AT	//	A	B	114	AX	BX	/
0	6	9	11	16	21	29	30	31

VSX Vector bfloat16 GER (rank-2 update) Negative multiply, Negative accumulate XX3-form

xvbf16ger2nn AT,XA,XB

59	AT	//	A	B	242	AX	BX	/
0	6	9	11	16	21	29	30	31

Prefixed Masked VSX Vector bfloat16 GER (rank-2 update) MMIRR:XX3-form

pmxvbf16ger2 AT,XA,XB,XMSK,YMSK,PMSK

Prefix:

1	3	9	//	/	/	PMSK	///	XMSK	YMSK
0	6	8	12	14	15	16	18	24	28 31

Suffix:

59	AT	//	A	B	51	AX	BX	/
0	6	9	11	16	21	29	30	31

Prefixed Masked VSX Vector bfloat16 GER (rank-2 update) Positive multiply, Positive accumulate MMIRR:XX3-form

pmxvbf16ger2pp AT,XA,XB,XMSK,YMSK,PMSK

Prefix:

1	3	9	//	/	/	PMSK	///	XMSK	YMSK
0	6	8	12	14	15	16	18	24	28 31

Suffix:

59	AT	//	A	B	50	AX	BX	/
0	6	9	11	16	21	29	30	31

Prefixed Masked VSX Vector bfloat16 GER (rank-2 update) Positive multiply, Negative accumulate MMIRR:XX3-form

pmxvbf16ger2pn AT,XA,XB,XMSK,YMSK,PMSK

Prefix:

1	3	9	//	/	/	PMSK	///	XMSK	YMSK
0	6	8	12	14	15	16	18	24	28 31

Suffix:

59	AT	//	A	B	178	AX	BX	/
0	6	9	11	16	21	29	30	31

Prefixed Masked VSX Vector bfloat16 GER (rank-2 update) Negative multiply, Positive accumulate MMIRR:XX3-form

pmxvbf16ger2np AT,XA,XB,XMSK,YMSK,PMSK

Prefix:

1	3	9	//	/	/	PMSK	///	XMSK	YMSK
0	6	8	12	14	15	16	18	24	28 31

Suffix:

59	AT	//	A	B	114	AX	BX	/
0	6	9	11	16	21	29	30	31

Prefixed Masked VSX Vector bfloat16 GER (rank-2 update) Negative multiply, Negative accumulate MMIRR:XX3-form

pmxvbf16ger2nn AT,XA,XB,XMSK,YMSK,PMSK

Prefix:

1	3	9	//	/	/	PMSK	///	XMSK	YMSK
0	6	8	12	14	15	16	18	24	28 31

Suffix:

59	AT	//	A	B	242	AX	BX	/
0	6	9	11	16	21	29	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

if ``xvbf16ger2'' | ``xvbf16ger2pp'' | ``xvbf16ger2pn'' | ``xvbf16ger2np'' | ``xvbf16ger2nn'' then
do
   PMSK ← 0b11      // enable all rank updates
   XMSK ← 0b1111    // enable all ACC[AT] rows
   YMSK ← 0b1111    // enable all ACC[AT] columns
end

do i = 0 to 3
   do j = 0 to 3
      if XMSK.bit[i]=1 & YMSK.bit[j]=1 then do
         src11 ← (PMSK.bit[0]=0) ? bfp_ZERO :
                       bfp_CONVERT_FROM_BFLOAT16(VSR[32×AX+A].word[i].hword[0])
         src21 ← (PMSK.bit[0]=0) ? bfp_ZERO :
                       bfp_CONVERT_FROM_BFLOAT16(VSR[32×BX+B].word[j].hword[0])
         src12 ← (PMSK.bit[1]=0) ? bfp_ZERO :
                       bfp_CONVERT_FROM_BFLOAT16(VSR[32×AX+A].word[i].hword[1])
         src22 ← (PMSK.bit[1]=0) ? bfp_ZERO :
                       bfp_CONVERT_FROM_BFLOAT16(VSR[32×BX+B].word[j].hword[1])

         reset_flags()

         p1 ← bfp_MULTIPLY(src11, src21)
         v1 ← bfp_MULTIPLY_ADD(src12, src22, p1)
         r1 ← bfp_ROUND_TO_BFP32_SIGNIFICAND(v1)

         if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
         if vximz_flag=1 then SetFX(FPSCR.VXIMZ)
         if vxisi_flag=1 then SetFX(FPSCR.VXISI)
         if xx_flag=1 then SetFX(FPSCR.XX)

         if ``[pm]xvbf16ger2'' then do
            reset_flags()

            r2 ← bfp_ROUND_TO_BFP32_DEFAULT(FPSCR.RN,r1)
            ACC[AT][i].word[j] ← bfp32_CONVERT_FROM_BFP(r2)

            if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
            if vxisi_flag=1 then SetFX(FPSCR.VXISI)
            if ox_flag=1 then SetFX(FPSCR.OX)
            if ux_flag=1 then SetFX(FPSCR.UX)
            if xx_flag=1 then SetFX(FPSCR.XX)
         end

         else do
            acc ← bfp_CONVERT_FROM_BFP32(ACC[AT][i].word[j])

            reset_flags()
```

```
            if ``[pm]xvbf16ger2pp'' then v ← bfp_ADD(r1, acc)
            if ``[pm]xvbf16ger2pn'' then v ← bfp_ADD(r1, bfp_NEGATE(acc))
            if ``[pm]xvbf16ger2np'' then v ← bfp_ADD(bfp_NEGATE(r1), acc)
            if ``[pm]xvbf16ger2nn'' then v ← bfp_ADD(bfp_NEGATE(r1), bfp_NEGATE(acc))

            r2 ← bfp_ROUND_TO_BFP32_DEFAULT(FPSCR.RN,v)
            ACC[AT][i].word[j] ← bfp32_CONVERT_FROM_BFP(r2)

            if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
            if vxisi_flag=1 then SetFX(FPSCR.VXISI)
            if ox_flag=1 then SetFX(FPSCR.OX)
            if ux_flag=1 then SetFX(FPSCR.UX)
            if xx_flag=1 then SetFX(FPSCR.XX)
          end
        end
        else
          ACC[AT][i][j] ← 0x0000_0000
    end
  end
```

Let XA be the value of 32×AX + A. If XA is in the range 4×AT to 4×AT+3, the instruction form is invalid.
Let XB be the value of 32×BX + B. If XB is in the range 4×AT to 4×AT+3, the instruction form is invalid.

Let the contents of ACC[AT] be a 4×4 matrix of single-precision floating-point values.

For ***xvbf16ger2***, ***xvbf16ger2pp***, ***xvbf16ger2pn***, ***xvbf16ger2np***, or ***xvbf16ger2nn***, let PMSK=0b11, XMSK=0b1111, and YMSK=0b1111.

For each integer value i from 0 to 3, and each integer value j from 0 to 3, do the following.

If bit i of XMSK is equal to 1 and bit j of YMSK is equal to 1, do the following.

If bit 0 of PMSK is equal to 1, let src10 be the bfloat16 floating-point value in halfword 0 of word element i of VSR[XA] and let src20 be the bfloat16 floating-point value in halfword 0 of word element j of VSR[XB]. Otherwise, let src10 be the value 0.0 and let src20 be the value 0.0, causing the product of src10 and src20 to be 0.0.

If bit 1 of PMSK is equal to 1, let src11 be the bfloat16 floating-point value in halfword 1 of word element i of VSR[XA] and let src21 be the bfloat16 floating-point value in halfword 1 of word element j of VSR[XB]. Otherwise, let src11 be the value 0.0 and let src21 be the value 0.0, causing the product of src11 and src21 to be 0.0.

Let prod be the product of src10 and src20, having infinite precision and unbounded exponent range.

Let psum be the sum of the product, src11 multiplied by src21, and prod, having infinite precision and unbounded exponent range.

Let r1 be the value psum with its significand rounded to 24-bit precision using the rounding mode specified by RN, but retaining unbounded exponent range (i.e., cannot overflow or underflow).

For [***pm***]***xvbf16ger2***, do the following.

Let r2 be the value r1 rounded to 24-bit significand precision and 8-bit exponent range (i.e., single-precision) using the rounding mode specified by RN.

r2 is placed into word element j of ACC[AT][i] in single-precision floating-point format.

For [***pm***]***xvbf16ger2pp***, do the following.

Let v2 be the sum of r1 added to the single-precision floating-point value in word element j of ACC[AT][i], having infinite precision and unbounded exponent range.

Let r2 be the value v2 rounded to 24-bit significand precision and 8-bit exponent range (i.e., single-precision) using the rounding mode specified by RN.

r2 is placed into word element j of ACC[AT][i] in single-precision floating-point format.

For [***pm***]***xvbf16ger2pn***, do the following.

Let v2 be the sum of r2 added to the negation of the single-precision floating-point value in word element j of ACC[AT][i], having infinite precision and unbounded exponent range.

Let r2 be the value v2 rounded to 24-bit significand precision and 8-bit exponent range (i.e., single-precision) using the rounding mode specified by RN.

r2 is placed into word element j of ACC[AT][i] in single-precision floating-point format.

For [***pm***]***xvbf16ger2np***, do the following.

Let v2 be the sum of the negation of r2 added to the single-precision floating-point value in word element j of ACC[AT][i], having infinite precision and unbounded exponent range.

Let r3 be the value v3 rounded to 24-bit significand precision and 8-bit exponent range (i.e., single-precision) using the rounding mode specified by RN.

r2 is placed into word element j of ACC[AT][i] in single-precision floating-point format.

For [***pm***]***xvbf16ger2nn***, do the following.

Let v2 be the sum of the negation of r2 added to the negation of the single-precision floating-point value in word element j of ACC[AT][i], having infinite precision and unbounded exponent range.

Let r2 be the value v3 rounded to 24-bit significand precision and 8-bit exponent range (i.e., single-precision) using the rounding mode specified by RN.

r2 is placed into word element j of ACC[AT][i] in single-precision floating-point format.

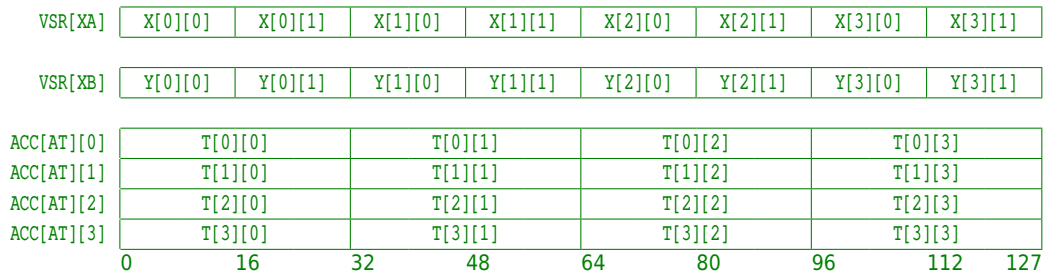
Otherwise, the contents of word element j of ACC[AT][i] are set to 0x0000_0000.

Unlike most other *VSX Vector Floating-Point* instructions, this instruction always updates the target register (here, ACC[AT], even when a trap-enabled exception occurs. For every multiply-add operation that is performed as part of the execution of this instruction, the operation is performed as if all exception enable bits are zero, and the trap-disabled result is returned. If the operation causes underflow or produces an inexact result, whether the operation causes an Underflow or Inexact exception is based on the actual contents of the Underflow and Overflow Enable bits. If any operand is a Quiet NaN or Signalling NaN, the result is a Quiet NaN. Exception status is accumulated and the appropriate exception status bits in the FPSCR are updated at the completion of execution of the instruction. Otherwise, behavior is the same as any vector floating-point instruction that can cause an exception. Taking a Program interrupt on a trap-enabled exception when interrupts are enabled by MSR.FE0 and MSR.FE1 is still supported, albeit with ACC[AT] updated based on a trap-disabled result.

Special Registers Altered:

Register	***Field(s)***
FPSCR	FX VXSNAN VXIMZ VXISI OX UX XX

Register Operand Data Layout for [pm]xvbf16ger2[pp|pn|np|nn]



Programming Note

Let X be the 4×2 matrix of bfloat16 floating-point values contained in VSR[XA] in row-major format.
Let Y be the 4×2 matrix of bfloat16 floating-point values contained in VSR[XB] in row-major format.
Let ACC[AT] be the accumulator containing a 4×4 matrix of single-precision floating-point values.

[***pm***]***xvbf16ger2*** performs the following form of accumulation of two outer products (rank 2 update).

```
for i=0 to 3, j=0 to 3:
    ACC[AT][i][j] = fmadds(X[i][1],Y[j][1],fmulsx(X[i][0],Y[j][0]))
```

where fmulsx() is equivalent to a ***fmuls*** instruction that rounds the significand of its result to single-precision but retains an exponent having unbounded range.

[***pm***]***xvbf16ger2pp*** performs the following form of accumulation of two outer products (rank 2 update).

```
for i=0 to 3, j=0 to 3:
    ACC[AT][i][j] = fadds(fmadds(X[i][1],Y[j][1],fmulsx(X[i][0],Y[j][0])), ACC[AT][i][j])
```

[***pm***]***xvbf16ger2pn*** performs the following form of accumulation of two outer products (rank 2 update).

```
for i=0 to 3, j=0 to 3:
    ACC[AT][i][j] = fsubs(fmadds(X[i][1],Y[j][1],fmulsx(X[i][0],Y[j][0])), ACC[AT][i][j])
```

[***pm***]***xvbf16ger2np*** performs the following form of accumulation of two outer products (rank 2 update).

```
for i=0 to 3, j=0 to 3:
    ACC[AT][i][j] = fadds(fnmadds(X[i][1],Y[j][1],fmulsx(X[i][0],Y[j][0])), ACC[AT][i][j])
```

[***pm***]***xvbf16ger2nn*** performs the following form of accumulation of two outer products (rank 2 update).

```
for i=0 to 3, j=0 to 3:
    ACC[AT][i][j] = fsubs(fnmadds(X[i][1],Y[j][1],fmulsx(X[i][0],Y[j][0])), ACC[AT][i][j])
```

VSX Vector 16-bit Floating-Point GER (rank-2 update) XX3-form

xvf16ger2 AT,XA,XB

59	AT	//	A	B	19	AX	BX	/
0	6	9	11	16	21	29	30	31

VSX Vector 16-bit Floating-Point GER (rank-2 update) Positive multiply, Positive accumulate XX3-form

xvf16ger2pp AT,XA,XB

59	AT	//	A	B	18	AX	BX	/
0	6	9	11	16	21	29	30	31

VSX Vector 16-bit Floating-Point GER (rank-2 update) Positive multiply, Negative accumulate XX3-form

xvf16ger2pn AT,XA,XB

59	AT	//	A	B	146	AX	BX	/
0	6	9	11	16	21	29	30	31

VSX Vector 16-bit Floating-Point GER (rank-2 update) Negative multiply, Positive accumulate XX3-form

xvf16ger2np AT,XA,XB

59	AT	//	A	B	82	AX	BX	/
0	6	9	11	16	21	29	30	31

VSX Vector 16-bit Floating-Point GER (rank-2 update) Negative multiply, Negative accumulate XX3-form

xvf16ger2nn AT,XA,XB

59	AT	//	A	B	210	AX	BX	/
0	6	9	11	16	21	29	30	31

Prefixed Masked VSX Vector 16-bit Floating-Point GER (rank-2 update) MMIRR:XX3-form

pmxvf16ger2 AT,XA,XB,XMSK,YMSK,PMSK

Prefix:

1	3	9	//	/	/	PMSK	///	XMSK	YMSK
0	6	8	12	14	15	16	18	24	28 31

Suffix:

59	AT	//	A	B	19	AX	BX	/
0	6	9	11	16	21	29	30	31

Prefixed Masked VSX Vector 16-bit Floating-Point GER (rank-2 update) Positive multiply, Positive accumulate MMIRR:XX3-form

pmxvf16ger2pp AT,XA,XB,XMSK,YMSK,PMSK

Prefix:

1	3	9	//	/	/	PMSK	///	XMSK	YMSK
0	6	8	12	14	15	16	18	24	28 31

Suffix:

59	AT	//	A	B	18	AX	BX	/
0	6	9	11	16	21	29	30	31

Prefixed Masked VSX Vector 16-bit Floating-Point GER (rank-2 update) Positive multiply, Negative accumulate MMIRR:XX3-form

pmxvf16ger2pn AT,XA,XB,XMSK,YMSK,PMSK

Prefix:

1	3	9	//	/	/	PMSK	///	XMSK	YMSK
0	6	8	12	14	15	16	18	24	28 31

Suffix:

59	AT	//	A	B	146	AX	BX	/
0	6	9	11	16	21	29	30	31

Prefixed Masked VSX Vector 16-bit Floating-Point GER (rank-2 update) Negative multiply, Positive accumulate MMIRR:XX3-form

pmxvf16ger2np AT,XA,XB,XMSK,YMSK,PMSK

Prefix:

Suffix:

Prefixed Masked VSX Vector 16-bit Floating-Point GER (rank-2 update) Negative multiply, Negative accumulate MMIRR:XX3-form

pmxvf16ger2nn AT,XA,XB,XMSK,YMSK,PMSK

Prefix:

Suffix:


```
if MSR.VSX=0 then VSX_Unavailable()

if ``xvf16ger2'' | ``xvf16ger2pp'' | ``xvf16ger2pn'' | ``xvf16ger2np'' | ``xvf16ger2nn'' then do
   PMSK ← 0b11
   XMSK ← 0b1111
   YMSK ← 0b1111
end

do i = 0 to 3
   do j = 0 to 3
      if XMSK.bit[i] & YMSK.bit[j] then do
         reset_flags()

         src10 ← bfp_CONVERT_FROM_BFP16((PMSK.bit[0]=0) ? 0x0000 : VSR[32×AX+A].word[i].hword[0])
         src11 ← bfp_CONVERT_FROM_BFP16((PMSK.bit[1]=0) ? 0x0000 : VSR[32×AX+A].word[i].hword[1])
         src20 ← bfp_CONVERT_FROM_BFP16((PMSK.bit[0]=0) ? 0x0000 : VSR[32×BX+B].word[j].hword[0])
         src21 ← bfp_CONVERT_FROM_BFP16((PMSK.bit[1]=0) ? 0x0000 : VSR[32×BX+B].word[j].hword[1])

         p1    ← bfp_MULTIPLY(src10, src20)
         v1    ← bfp_MULTIPLY_ADD(src11, src21, p1)
         r1    ← bfp_ROUND_TO_BFP32_DEFAULT(FPSCR.RN, v1)

         if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
         if vximz_flag=1 then SetFX(FPSCR.VXIMZ)
         if vxisi_flag=1 then SetFX(FPSCR.VXISI)
         if ox_flag=1 then SetFX(FPSCR.OX)
         if ux_flag=1 then SetFX(FPSCR.UX)
         if xx_flag=1 then SetFX(FPSCR.XX)

         reset_flags()

         if ``[pm]xvf16ger2'' then
            ACC[AT][i].word[j] ← bfp32_CONVERT_FROM_BFP(r1)

         else do
            acc ← bfp_CONVERT_FROM_BFP32(ACC[AT][i].word[j])

            if ``[pm]xvf16ger2pp'' then v2 ← bfp_ADD(r1, acc)
            if ``[pm]xvf16ger2pn'' then v2 ← bfp_ADD(r1, bfp_NEGATE(acc))
            if ``[pm]xvf16ger2np'' then v2 ← bfp_ADD(bfp_NEGATE(r1), acc)
            if ``[pm]xvf16ger2nn'' then v2 ← bfp_ADD(bfp_NEGATE(r1), bfp_NEGATE(acc))

            r2 ← bfp_ROUND_TO_BFP32_DEFAULT(FPSCR.RN, v2)
```

```
            ACC[AT][i].word[j] ← bfp32_CONVERT_FROM_BFP(r2)

            if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
            if vxisi_flag=1 then SetFX(FPSCR.VXISI)
            if ox_flag=1 then SetFX(FPSCR.OX)
            if ux_flag=1 then SetFX(FPSCR.UX)
            if xx_flag=1 then SetFX(FPSCR.XX)
         end
      end
      else
         ACC[AT][i][j] ← 0x0000_0000
   end
end
```

Let XA be the value of 32×AX + A. If XA is in the range 4×AT to 4×AT+3, the instruction form is invalid.
Let XB be the value of 32×BX + B. If XB is in the range 4×AT to 4×AT+3, the instruction form is invalid.

Let the contents of ACC[AT] be a 4×4 matrix of single-precision floating-point values.

Let result be a 4×4 matrix of word elements to be used as a temporary Accumulator.

For ***xvf16ger2***, ***xvf16ger2pp***, ***xvf16ger2pn***, ***xvf16ger2np***, or ***xvf16ger2nn***, let PMSK=0b11, XMSK=0b1111, and YMSK=0b1111.

For each integer value i from 0 to 3, and each integer value j from 0 to 3, do the following.

If bit i of XMSK is equal to 1 and bit j of YMSK is equal to 1, do the following.

If bit 0 of PMSK is equal to 1, let src10 be the half-precision floating-point value in hword 0 of word element i of VSR[XA] and let src20 be the half-precision floating-point value in hword 0 of word element j of VSR[XB]. Otherwise, let src10 be the value 0.0 and let src20 be the value 0.0, causing the product of src10 and src20 to be 0.0.

If bit 1 of PMSK is equal to 1, let src11 be the half-precision floating-point value in hword 1 of word element i of VSR[XA] and let src21 be the half-precision floating-point value in hword 1 of word element j of VSR[XB]. Otherwise, let src11 be the value 0.0 and let src21 be the value 0.0, causing the product of src11 and src21 to be 0.0.

Let prod be the single-precision product of src10 and src20.

Let msum be the sum of prod added to the product of src11 and src21. msum is rounded to single-precision using the rounding mode specified in RN.

For [***pm***]***xvf16ger2***, the rounded msum is placed into word element j of ACC[AT][i] in single-precision floating-point format.

For [***pm***]***xvf16ger2pp***, the rounded msum is added to the single-precision floating-point value in word element j of ACC[AT][i], rounded to single-precision using the rounding mode specified in RN, and placed into word element j of ACC[AT][i] in single-precision floating-point format.

For [***pm***]***xvf16ger2pn***, the rounded msum is added to the negation of the single-precision floating-point value in word element j of ACC[AT][i], rounded to single-precision using the rounding mode specified in RN, and placed into word element j of ACC[AT][i] in single-precision floating-point format.

For [***pm***]***xvf16ger2np***, the negation of the rounded msum is added to the single-precision floating-point value in word element j of ACC[AT][i], rounded to single-precision using the rounding mode specified in RN, and placed into word element j of ACC[AT][i] in single-precision floating-point format.

For [***pm***]***xvf16ger2nn***, the negation of the rounded msum is added to the negation of the single-precision floating-point value in word element j of ACC[AT][i], rounded to single-precision using the rounding mode specified in RN, and placed into word element j of ACC[AT][i] in single-precision floating-point format.

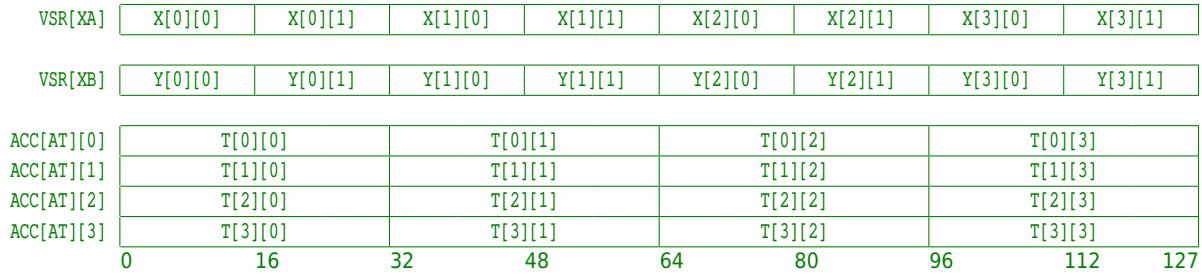
Otherwise, the contents of ACC[AT][i][j] are set to 0x0000_0000.

Unlike most other *VSX Vector Floating-Point* instructions, this instruction always updates the target register (here, ACC[AT], even when a trap-enabled exception occurs. For every multiply-add operation that is performed as part of the execution of this instruction, the operation is performed as if all exception enable bits are zero, and the trap-disabled result is returned. If the operation causes underflow or produces an inexact result, whether the operation causes an Underflow or Inexact exception is based on the actual contents of the Underflow and Overflow Enable bits. If any operand is a Quiet NaN or Signalling NaN, the result is a Quiet NaN. Exception status is accumulated and the appropriate exception status bits in the FPSCR are updated at the completion of execution of the instruction. Otherwise, behavior is the same as any vector floating-point instruction that can cause an exception. Taking a Program interrupt on a trap-enabled exception when interrupts are enabled by `MSR.FE0` and `MSR.FE1` is still supported, albeit with `ACC[AT]` updated based on a trap-disabled result.

Special Registers Altered:

Register	***Field(s)***	
FPSCR	FX VXSNAN VXIMZ OX UX XX	(if [***pm***]***xvf16ger2***)
FPSCR	FX VXSNAN VXIMZ VXISI OX UX XX	(if [***pm***]***xvf16ger2pp***)
FPSCR	FX VXSNAN VXIMZ VXISI OX UX XX	(if [***pm***]***xvf16ger2pn***)
FPSCR	FX VXSNAN VXIMZ VXISI OX UX XX	(if [***pm***]***xvf16ger2np***)
FPSCR	FX VXSNAN VXIMZ VXISI OX UX XX	(if [***pm***]***xvf16ger2nn***)

Register Operand Data Layout for [pm]xvf16ger2[pp|pn|np|nn]



Programming Note

Let X be the 4×2 matrix of half-precision floating-point values contained in VSR[XA] in row-major format.
Let Y be the 4×2 matrix of half-precision floating-point values contained in VSR[XB] in row-major format.
Let ACC[AT] be the Accumulator containing a 4×4 matrix of single-precision floating-point values.

Note that floating-point arithmetic is not associative. That is, (X+Y)+T can return a different result than X+(Y+T). The ordering specified by the instruction description for any result element i,j is the order the operations will be performed in hardware. The floating-point operations to implement each result element for ***xvf16ger2***[*pp*|*pn*|*np*|*nn*] are shown below.

[***pm***]***xvf16ger2*** performs the following form of accumulation of two outer products (rank 2 update).

```
for i=0 to 3, j=0 to 3:
    ACC[AT][i][j] = fmadds(X[i][1],Y[j][1],fmuls(X[i][0],Y[j][0]))
```

[***pm***]***xvf16ger2pp*** performs the following form of accumulation of two outer products (rank 2 update).

```
for i=0 to 3, j=0 to 3:
    ACC[AT][i][j] = fadds(fmadds(X[i][1],Y[j][1],fmuls(X[i][0],Y[j][0])),ACC[AT][i][j])
```

[***pm***]***xvf16ger2pn*** performs the following form of accumulation of two outer products (rank 2 update).

```
for i=0 to 3, j=0 to 3:
    ACC[AT][i][j] = fsubs(fmadds(X[i][1],Y[j][1],fmuls(X[i][0],Y[j][0])),ACC[AT][i][j])
```

[***pm***]***xvf16ger2np*** performs the following form of accumulation of two outer products (rank 2 update).

```
for i=0 to 3, j=0 to 3:
    ACC[AT][i][j] = fadds(fnmadds(X[i][1],Y[j][1],fmuls(X[i][0],Y[j][0])),ACC[AT][i][j])
```

[***pm***]***xvf16ger2nn*** performs the following form of accumulation of two outer products (rank 2 update).

```
for i=0 to 3, j=0 to 3:
    ACC[AT][i][j] = fsubs(fnmadds(X[i][1],Y[j][1],fmuls(X[i][0],Y[j][0])),ACC[AT][i][j])
```

VSX Vector 32-bit Floating-Point GER (rank-1 update) XX3-form

xvf32ger AT,XA,XB

59	AT	//	A	B	27	AX	BX	/
0	6	9	11	16	21	29	30	31

VSX Vector 32-bit Floating-Point GER (rank-1 update) Positive multiply, Positive accumulate XX3-form

xvf32gerpp AT,XA,XB

59	AT	//	A	B	26	AX	BX	/
0	6	9	11	16	21	29	30	31

VSX Vector 32-bit Floating-Point GER (rank-1 update) Positive multiply, Negative accumulate XX3-form

xvf32gerpn AT,XA,XB

59	AT	//	A	B	154	AX	BX	/
0	6	9	11	16	21	29	30	31

VSX Vector 32-bit Floating-Point GER (rank-1 update) Negative multiply, Positive accumulate XX3-form

xvf32gernp AT,XA,XB

59	AT	//	A	B	90	AX	BX	/
0	6	9	11	16	21	29	30	31

VSX Vector 32-bit Floating-Point GER (rank-1 update) Negative multiply, Negative accumulate XX3-form

xvf32gernn AT,XA,XB

59	AT	//	A	B	218	AX	BX	/
0	6	9	11	16	21	29	30	31

Prefixed Masked VSX Vector 32-bit Floating-Point GER (rank-1 update) MMIRR:XX3-form

pmxvf32ger AT,XA,XB,XMSK,YMSK

Prefix:

1	3	9	//	/	/	///	XMSK	YMSK
0	6	8	12	14	15	16	24	28 31

Suffix:

59	AT	//	A	B	27	AX	BX	/
0	6	9	11	16	21	29	30	31

Prefixed Masked VSX Vector 32-bit Floating-Point GER (rank-1 update) Positive multiply, Positive accumulate MMIRR:XX3-form

pmxvf32gerpp AT,XA,XB,XMSK,YMSK

Prefix:

1	3	9	//	/	/	///	XMSK	YMSK
0	6	8	12	14	15	16	24	28 31

Suffix:

59	AT	//	A	B	26	AX	BX	/
0	6	9	11	16	21	29	30	31

Prefixed Masked VSX Vector 32-bit Floating-Point GER (rank-1 update) Positive multiply, Negative accumulate MMIRR:XX3-form

pmxvf32gerpn AT,XA,XB,XMSK,YMSK

Prefix:

1	3	9	//	/	/	///	XMSK	YMSK
0	6	8	12	14	15	16	24	28 31

Suffix:

59	AT	//	A	B	154	AX	BX	/
0	6	9	11	16	21	29	30	31

Prefixed Masked VSX Vector 32-bit Floating-Point GER (rank-1 update) Negative multiply, Positive accumulate MMIRR:XX3-form

pmxvf32gernp AT,XA,XB,XMSK,YMSK

Prefix:

1	3	9	//	/	/	///	XMSK	YMSK
0	6	8	12	14	15	16	24	28 31

Suffix:

59	AT	//	A	B	90	AX	BX	/
0	6	9	11	16	21	29	30	31

Prefixed Masked VSX Vector 32-bit Floating-Point GER (rank-1 update) Negative multiply, Negative accumulate MMIRR:XX3-form

pmxvf32gernn AT,XA,XB,XMSK,YMSK

Prefix:

1	3	9	//	/	/	///	XMSK	YMSK
0	6	8	12	14	15	16	24	28 31

Suffix:

59	AT	//	A	B	218	AX	BX	/
0	6	9	11	16	21	29	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

if ''xvf32ger'' | ''xvf32gerpp'' | ''xvf32gerpn'' | ''xvf32gernp'' | ''xvf32gernn'' then do
   XMSK ← 0b1111
   YMSK ← 0b1111
end

do i = 0 to 3
   do j = 0 to 3
      if XMSK.bit[i]=1 & YMSK.bit[j]=1 then do
         reset_flags()

         src1 ← bfp_CONVERT_FROM_BFP32(VSR[32×AX+A].word[i])
         src2 ← bfp_CONVERT_FROM_BFP32(VSR[32×BX+B].word[j])
         acc  ← bfp_CONVERT_FROM_BFP32(ACC[AT][i].word[j])

         if ''[pm]xvf32ger''   then v ← bfp_MULTIPLY( src1, src2 )
         if ''[pm]xvf32gerpp'' then v ← bfp_MULTIPLY_ADD( src1, src2, acc )
         if ''[pm]xvf32gerpn'' then v ← bfp_MULTIPLY_ADD( src1, src2, bfp_NEGATE(acc) )
         if ''[pm]xvf32gernp'' then v ← bfp_MULTIPLY_ADD( src1, src2, bfp_NEGATE(acc) )
         if ''[pm]xvf32gernn'' then v ← bfp_MULTIPLY_ADD( src1, src2, acc )

         r ← bfp_ROUND_TO_BFP32_DEFAULT(FPSCR.RN,v)

         if ''[pm]xvf32gernp'' then r ← bfp_NEGATE(r)
         if ''[pm]xvf32gernn'' then r ← bfp_NEGATE(r)

         ACC[AT][i].word[j] ← bfp32_CONVERT_FROM_BFP(r)

         if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
         if vximz_flag=1 then SetFX(FPSCR.VXIMZ)
         if vxisi_flag=1 then SetFX(FPSCR.VXISI)
         if ox_flag=1 then SetFX(FPSCR.OX)
         if ux_flag=1 then SetFX(FPSCR.UX)
         if xx_flag=1 then SetFX(FPSCR.XX)
      end
      else
         ACC[AT][i].word[j] ← 0x0000_0000
   end
end
```

Let XA be the value of 32×AX + A. If XA is in the range 4×AT to 4×AT+3, the instruction form is invalid.
Let XB be the value of 32×BX + B. If XB is in the range 4×AT to 4×AT+3, the instruction form is invalid.

Let the contents of ACC[AT] be a 4×4 matrix of single-precision floating-point values.

For ***xvf32ger***, ***xvf32gerpp***, ***xvf32gerpn***, ***xvf32gernp***, or ***xvf32gernn***, let XMSK=0b1111 and YMSK=0b1111.

For each integer value i from 0 to 3, and each integer value j from 0 to 3, do the following.

If bit i of XMSK is equal to 1 and bit j of YMSK is equal to 1, do the following.

Let prod be the product of the single-precision floating-point value in word element i of VSR[XA] and the single-precision floating-point value in word element j of VSR[XB], having unbounded range and precision.

For [***pm***]***xvf32ger***, prod is rounded to single-precision using the rounding mode specified in RN. The rounded result is placed into word element j of ACC[AT][i] in single-precision floating-point format.

For [***pm***]***xvf32gerpp***, the single-precision floating-point value in word element j of ACC[AT][i] is added to prod. The intermediate result is rounded to single-precision using the rounding mode specified in RN. The rounded result is placed into word element j of ACC[AT][i] in single-precision floating-point format.

For [***pm***]***xvf32gerpn***, the single-precision floating-point value in word element j of ACC[AT][i] is subtracted from prod. The intermediate result is rounded to single-precision using the rounding mode specified in RN. The rounded result is placed into word element j of ACC[AT][i] in single-precision floating-point format.

For [***pm***]***xvf32gernp***, the single-precision floating-point value in word element j of ACC[AT][i] is subtracted from prod. The intermediate result is rounded to single-precision using the rounding mode specified in RN. The rounded result is negated and placed into word element j of ACC[AT][i] in single-precision floating-point format.

For [***pm***]***xvf32gernn***, the single-precision floating-point value in word element j of ACC[AT][i] is added to prod. The intermediate result is rounded to single-precision using the rounding mode specified in RN. The rounded result is negated and placed into word element j of ACC[AT][i] in single-precision floating-point format.

Otherwise, the contents of word element j of ACC[AT][i] are set to 0x0000_0000.

Unlike other vector floating-point instructions, ACC[AT] is always updated by the execution of the instruction, even when a trap-enabled exception occurs. For every multiply-add operation that is performed as part of the execution of this instruction, if an exception occurs as the result of that particular multiply-add operation, the trap-disabled exception result is returned, even if that exception type is trap-enabled. If any operand is a Quiet NaN or Signalling NaN, the result is a Quiet NaN. Exception detection is based on the trap-disabled definition. Exception status is accumulated and the appropriate exception status bits in the FPSCR are updated at the completion of execution of the instruction. Otherwise, behavior is the same as any vector floating-point instruction that can cause an exception. Taking a Program interrupt on a trap-enabled exception when interrupts are enabled by MSR.FE0 and MSR.FE1 is still supported, albeit with the ACC[AT] updated based on a trap-disabled result.

Special Registers Altered:

Register	***Field(s)***	
FPSCR	FX VXSNAN VXIMZ OX UX XX	(if [***pm***]***xvf32ger***)
FPSCR	FX VXSNAN VXIMZ VXISI OX UX XX	(if [***pm***]***xvf32gerpp***)
FPSCR	FX VXSNAN VXIMZ VXISI OX UX XX	(if [***pm***]***xvf32gerpn***)
FPSCR	FX VXSNAN VXIMZ VXISI OX UX XX	(if [***pm***]***xvf32gernp***)
FPSCR	FX VXSNAN VXIMZ VXISI OX UX XX	(if [***pm***]***xvf32gernn***)

Register Operand Data Layout for [pm]xvf32ger[pp|pn|np|nn]

VSR[XA]	X[0]	X[1]	X[2]	X[3]
VSR[XB]	Y[0]	Y[1]	Y[2]	Y[3]
ACC[AT][0]	T[0][0]	T[0][1]	T[0][2]	T[0][3]
ACC[AT][1]	T[1][0]	T[1][1]	T[1][2]	T[1][3]
ACC[AT][2]	T[2][0]	T[2][1]	T[2][2]	T[2][3]
ACC[AT][3]	T[3][0]	T[3][1]	T[3][2]	T[3][3]

Programming Note

Let X be the 4–element vector of single-precision floating-point values contained in VSR[XA].
Let Y be the 4–element vector of single-precision floating-point values contained in VSR[XB].
Let ACC[AT] be the Accumulator containing a 4×4 matrix of single-precision floating-point values.

The floating-point operations to implement each result element for ***xvf32ger***[*pp*|*pn*|*np*|*nn*] are shown below.

[***pm***]***xvf32ger*** performs the following form of accumulation of one outer product (rank 1 update).

```
for i=0 to 3, j=0 to 3:
    ACC[AT][i][j] = fmuls(X[i],Y[j])
```

[***pm***]***xvf32gerpp*** performs the following form of accumulation of one outer product (rank 1 update).

```
for i=0 to 3, j=0 to 3:
    ACC[AT][i][j] = fmadds(X[i],Y[j],ACC[AT][i][j])
```

[***pm***]***xvf32gerpn*** performs the following form of accumulation of one outer product (rank 1 update).

```
for i=0 to 3, j=0 to 3:
    ACC[AT][i][j] = fmsubs(X[i],Y[j],ACC[AT][i][j])
```

[***pm***]***xvf32gernp*** performs the following form of accumulation of one outer product (rank 1 update).

```
for i=0 to 3, j=0 to 3:
    ACC[AT][i][j] = fnmsubs(X[i],Y[j],ACC[AT][i][j])
```

[***pm***]***xvf32gernn*** performs the following form of accumulation of one outer product (rank 1 update).

```
for i=0 to 3, j=0 to 3:
    ACC[AT][i][j] = fnmadds(X[i],Y[j],ACC[AT][i][j])
```

VSX Vector 64-bit Floating-Point GER (rank-1 update) XX3-form

xvf64ger AT,XAp,XB

59	AT	//	Ap	B	59	AX	BX	/
0	6	9	11	16	21	29	30	31

VSX Vector 64-bit Floating-Point GER (rank-1 update) Positive multiply, Positive accumulate XX3-form

xvf64gerpp AT,XAp,XB

59	AT	//	Ap	B	58	AX	BX	/
0	6	9	11	16	21	29	30	31

VSX Vector 64-bit Floating-Point GER (rank-1 update) Positive multiply, Negative accumulate XX3-form

xvf64gerpn AT,XAp,XB

59	AT	//	Ap	B	186	AX	BX	/
0	6	9	11	16	21	29	30	31

VSX Vector 64-bit Floating-Point GER (rank-1 update) Negative multiply, Positive accumulate XX3-form

xvf64gernp AT,XAp,XB

59	AT	//	Ap	B	122	AX	BX	/
0	6	9	11	16	21	29	30	31

VSX Vector 64-bit Floating-Point GER (rank-1 update) Negative multiply, Negative accumulate XX3-form

xvf64gernn AT,XAp,XB

59	AT	//	Ap	B	250	AX	BX	/
0	6	9	11	16	21	29	30	31

Prefixed Masked VSX Vector 64-bit Floating-Point GER (rank-1 update) MMIRR:XX3-form

pmxvf64ger AT,XAp,XB,XMSK,YMSK

Prefix:

1	3	9	//	/	/	///	XMSK	YMSK	
0	6	8	12	14	15	16	24	28	30 31

Suffix:

59	AT	//	Ap	B	59	AX	BX	/
0	6	9	11	16	21	29	30	31

Prefixed Masked VSX Vector 64-bit Floating-Point GER (rank-1 update) Positive multiply, Positive accumulate MMIRR:XX3-form

pmxvf64gerpp AT,XAp,XB,XMSK,YMSK

Prefix:

1	3	9	//	/	/	///	XMSK	YMSK	
0	6	8	12	14	15	16	24	28	30 31

Suffix:

59	AT	//	Ap	B	58	AX	BX	/
0	6	9	11	16	21	29	30	31

Prefixed Masked VSX Vector 64-bit Floating-Point GER (rank-1 update) Positive multiply, Negative accumulate MMIRR:XX3-form

pmxvf64gerpn AT,XAp,XB,XMSK,YMSK

Prefix:

1	3	9	//	/	/	///	XMSK	YMSK	
0	6	8	12	14	15	16	24	28	30 31

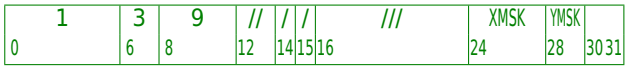
Suffix:

59	AT	//	Ap	B	186	AX	BX	/
0	6	9	11	16	21	29	30	31

Prefixed Masked VSX Vector 64-bit Floating-Point GER (rank-1 update) Negative multiply, Positive accumulate MMIRR:XX3-form

pmxvf64gernp AT,XAp,XB,XMSK,YMSK

Prefix:

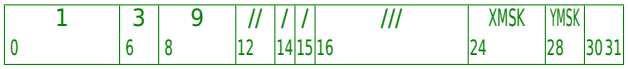


Suffix:

Prefixed Masked VSX Vector 64-bit Floating-Point GER (rank-1 update) Negative multiply, Negative accumulate MMIRR:XX3-form

pmxvf64gernn AT,XAp,XB,XMSK,YMSK

Prefix:



Suffix:


```
if MSR.VSX=0 then VSX_Unavailable()

if ''xvf64ger'' | ''xvf64gerpp'' | ''xvf64gerpn'' | ''xvf64gernp'' | ''xvf64gernn'' then do
   XMSK ← 0b1111
   YMSK ← 0b11
end
vsrc1.qword[0] ← VSR[32×AX+Ap]
vsrc1.qword[1] ← VSR[32×AX+Ap+1]
vsrc2          ← VSR[32×BX+B]
do i = 0 to 3
   do j = 0 to 1
      if XMSK.bit[i]=1 & YMSK.bit[j]=1 then do
         reset_flags()

         src1 ← bfp_CONVERT_FROM_BFP64(vsrc1.dword[i])
         src2 ← bfp_CONVERT_FROM_BFP64(vsrc2.dword[j])
         acc  ← bfp_CONVERT_FROM_BFP64(ACC[AT][i].dword[j])

         if ''[pm]xvf64ger''   then v ← bfp_MULTIPLY( src1, src2 )
         if ''[pm]xvf64gerpp'' then v ← bfp_MULTIPLY_ADD( src1, src2, acc )
         if ''[pm]xvf64gerpn'' then v ← bfp_MULTIPLY_ADD( src1, src2, bfp_NEGATE(acc) )
         if ''[pm]xvf64gernp'' then v ← bfp_MULTIPLY_ADD( src1, src2, bfp_NEGATE(acc) )
         if ''[pm]xvf64gernn'' then v ← bfp_MULTIPLY_ADD( src1, src2, acc )

         r ← bfp_ROUND_TO_BFP64_DEFAULT(FPSCR.RN,v)

         if ''[pm]xvf64gernp'' then r ← bfp_NEGATE(r)
         if ''[pm]xvf64gernn'' then r ← bfp_NEGATE(r)

         ACC[AT][i].dword[j] ← bfp64_CONVERT_FROM_BFP(r)

         if vxsnan_flag=1 then SetFX(FPSCR.VXSNAN)
         if vximz_flag=1 then SetFX(FPSCR.VXIMZ)
         if vxisi_flag=1 then SetFX(FPSCR.VXISI)
         if ox_flag=1 then SetFX(FPSCR.OX)
         if ux_flag=1 then SetFX(FPSCR.UX)
         if xx_flag=1 then SetFX(FPSCR.XX)
      end
      else
         ACC[AT][i].dword[j] ← 0x0000_0000_0000_0000
   end
end
```

Let XAp be the value of 32×AX + Ap. If XAp is odd, or is in the range 4×AT to 4×AT+3, the instruction form is invalid.
Let XB be the value of 32×BX + B. If XB is in the range 4×AT to 4×AT+3, the instruction form is invalid.

Let the contents of ACC[AT] be a 4×2 matrix of double-precision floating-point values.

Let vsrcX be the concatenation of the contents of VSR[XAp] and VSR[XAp+1].
Let vsrcY be the contents of VSR[XB].

For ***xvf64ger***, ***xvf64gerpp***, ***xvf64gerpn***, ***xvf64gernp***, and ***xvf64gernn***, let XMSK=0b1111 and YMSK=0b11.

For each integer value i from 0 to 3, and each integer value j from 0 to 1, do the following.

If bit i of XMSK is equal to 1 and bit j of YMSK is equal to 1, do the following.

Let prod be the product of the double-precision floating-point value in doubleword element i of vsrcX and the double-precision floating-point value in doubleword element j of vsrcY, having unbounded range and precision.

For [***pm***]***xvf64ger***, prod is rounded to double-precision using the rounding mode specified in RN. The rounded result is placed into doubleword element j of ACC[AT][i] in double-precision floating-point format.

For [***pm***]***xvf64gerpp***, prod is added to the double-precision floating-point value in doubleword element j of ACC[AT][i]. The intermediate result is rounded to double-precision using the rounding mode specified in RN. The rounded result is placed into doubleword element j of ACC[AT][i] in double-precision floating-point format.

For [***pm***]***xvf64gerpn***, prod is added to the negation of the double-precision floating-point value in doubleword element j of ACC[AT][i]. The intermediate result is rounded to double-precision using the rounding mode specified in RN. The rounded result is placed into doubleword element j of ACC[AT][i] in double-precision floating-point format.

For [***pm***]***xvf64gernn***, prod is added to the double-precision floating-point value in doubleword element j of ACC[AT][i]. The intermediate result is rounded to double-precision using the rounding mode specified in RN. The rounded result is negated and placed into doubleword element j of ACC[AT][i] in double-precision floating-point format.

For [***pm***]***xvf64gernp***, prod is added to the negation of the double-precision floating-point value in doubleword element j of ACC[AT][i]. The intermediate result is rounded to double-precision using the rounding mode specified in RN. The rounded result is negated and placed into doubleword element j of ACC[AT][i] in double-precision floating-point format.

Otherwise, the contents of doubleword element j of ACC[AT][i] are set to 0x0000_0000_0000_0000.

Unlike most other *VSX Vector Floating-Point* instructions, this instruction always updates the target register (here, ACC[AT], even when a trap-enabled exception occurs. For every multiply-add operation that is performed as part of the execution of this instruction, the operation is performed as if all exception enable bits are zero, and the trap-disabled result is returned. If the operation causes underflow or produces an inexact result, whether the operation causes an Underflow or Inexact exception is based on the actual contents of the Underflow and Overflow Enable bits. If any operand is a Quiet NaN or Signalling NaN, the result is a Quiet NaN. Exception status is accumulated and the appropriate exception status bits in the FPSCR are updated at the completion of execution of the instruction. Otherwise, behavior is the same as any vector floating-point instruction that can cause an exception. Taking a Program interrupt on a trap-enabled exception when interrupts are enabled by MSR.FE0 and MSR.FE1 is still supported, albeit with ACC[AT] updated based on a trap-disabled result.

Special Registers Altered:

Register	***Field(s)***	
FPSCR	FX VXSNAN VXIMZ OX UX XX	(if [***pm***]***xvf64ger***)
FPSCR	FX VXSNAN VXIMZ VXISI OX UX XX	(if [***pm***]***xvf64gerpp***)
FPSCR	FX VXSNAN VXIMZ VXISI OX UX XX	(if [***pm***]***xvf64gerpn***)
FPSCR	FX VXSNAN VXIMZ VXISI OX UX XX	(if [***pm***]***xvf64gernp***)
FPSCR	FX VXSNAN VXIMZ VXISI OX UX XX	(if [***pm***]***xvf64gernn***)

Register Operand Data Layout for [pm]xvf64ger[pp|pn|np|nn]

VSR[XAp]	X[0]	X[1]
VSR[XAp+1]	X[2]	X[3]

VSR[XB]	Y[0]	Y[1]

ACC[AT][0]	T[0][0]	T[0][1]
ACC[AT][1]	T[1][0]	T[1][1]
ACC[AT][2]	T[2][0]	T[2][1]
ACC[AT][3]	T[3][0]	T[3][1]

Programming Note

Let X be the 4–element vector of double-precision floating-point values contained in the concatenation of VSR[XAp] and VSR[XAp+1].
Let Y be the 2–element vector of double-precision floating-point values contained in VSR[XB].
Let ACC[AT] be the Accumulator containing a 4×2 matrix of double-precision floating-point values.

The floating-point operations to implement each result element for ***xvf64ger***[*pp*|*pn*|*np*|*nn*] are shown below.

[***pm***]***xvf64ger*** performs the following form of accumulation of one outer product (rank 1 update).

```
for i=0 to 3, j=0 to 1:
    ACC[AT][i][j] = fmul(X[i],Y[j])
```

[***pm***]***xvf64gerpp*** performs the following form of accumulation of one outer product (rank 1 update).

```
for i=0 to 3, j=0 to 1:
    ACC[AT][i][j] = fmadd(X[i],Y[j],ACC[AT][i][j])
```

[***pm***]***xvf64gerpn*** performs the following form of accumulation of one outer product (rank 1 update).

```
for i=0 to 3, j=0 to 1:
    ACC[AT][i][j] = fmsub(X[i],Y[j],ACC[AT][i][j])
```

[***pm***]***xvf64gernp*** performs the following form of accumulation of one outer product (rank 1 update).

```
for i=0 to 3, j=0 to 1:
    ACC[AT][i][j] = fnmsub(X[i],Y[j],ACC[AT][i][j])
```

[***pm***]***xvf64gernn*** performs the following form of accumulation of one outer product (rank 1 update).

```
for i=0 to 3, j=0 to 1:
    ACC[AT][i][j] = fnmadd(X[i],Y[j],ACC[AT][i][j])
```

7.6.2.13 VSX Vector Logical Instructions

7.6.2.13.1 VSX Vector Logical Instructions

VSX Vector Logical AND XX3-form

xxland XT,XA,XB

60	T	A	B	130	AX	BX	TX
0	6	11	16	21	29	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

VSR[32×TX+T] ← VSR[32×AX+A] & VSR[32×BX+B]
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

The contents of VSR[XA] are ANDed with the contents of VSR[XB] and the result is placed into VSR[XT].

Special Registers Altered:

None

VSX Vector Logical AND with Complement XX3-form

xxlandc XT,XA,XB

60	T	A	B	138	AX	BX	TX
0	6	11	16	21	29	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

VSR[32×TX+T] ← VSR[32×AX+A] & ¬VSR[32×BX+B]
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

The contents of VSR[XA] are ANDed with the complement of the contents of VSR[XB] and the result is placed into VSR[XT].

Special Registers Altered:

None

VSX Vector Logical Equivalence XX3-form

xxleqv XT,XA,XB

60	T	A	B	186	AX	BX	TX
0	6	11	16	21	29	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

VSR[32×TX+T] ← VSR[32×AX+A] ≡ VSR[32×BX+B]
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

The contents of VSR[XA] are exclusive-ORed with the contents of VSR[XB] and the complemented result is placed into VSR[XT].

Special Registers Altered:

None

VSX Vector Logical NAND XX3-form

xxlnand XT,XA,XB

60	T	A	B	178	AX	BX	TX
0	6	11	16	21	29	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

VSR[32×TX+T] ← ¬( VSR[32×AX+A] & VSR[32×BX+B] )
```

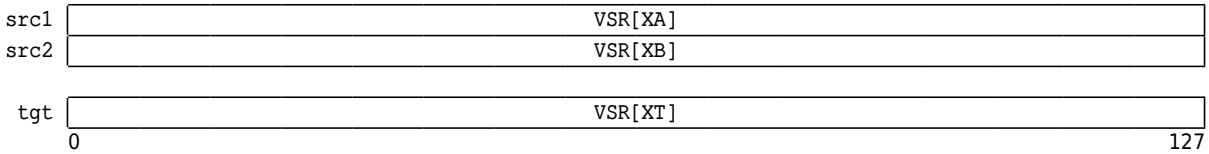
Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

The contents of VSR[XA] are ANDed with the contents of VSR[XB] and the complemented result is placed into VSR[XT].

Special Registers Altered:

None

VSR Data Layout for xxland, xxlandc, xxleqv, & xxlnand



VSX Vector Logical NOR XX3-form

xxlnor XT,XA,XB

60	T	A	B	162	AX	BX	TX
0	6	11	16	21	29	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

VSR[32×TX+T] ← ¬( VSR[32×AX+A] | VSR[32×BX+B] )
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

The contents of VSR[XA] are ORed with the contents of VSR[XB] and the complemented result is placed into VSR[XT].

Special Registers Altered:

None

VSX Vector Logical OR XX3-form

xxlor XT,XA,XB

60	T	A	B	146	AX	BX	TX
0	6	11	16	21	29	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

VSR[32×TX+T] ← VSR[32×AX+A] | VSR[32×BX+B]
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

The contents of VSR[XA] are ORed with the contents of VSR[XB] and the result is placed into VSR[XT].

Special Registers Altered:

None

VSX Vector Logical OR with Complement XX3-form

xxlorc XT,XA,XB

60	T	A	B	170	AX	BX	TX
0	6	11	16	21	29	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

VSR[32×TX+T] ← VSR[32×AX+A] | ¬VSR[32×BX+B]
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

The contents of VSR[XA] are ORed with the complement of the contents of VSR[XB] and the result is placed into VSR[XT].

Special Registers Altered:

None

VSX Vector Logical XOR XX3-form

xxlxor XT,XA,XB

60	T	A	B	154	AX	BX	TX
0	6	11	16	21	29	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

VSR[32×TX+T] ← VSR[32×AX+A] ⊕ VSR[32×BX+B]
```

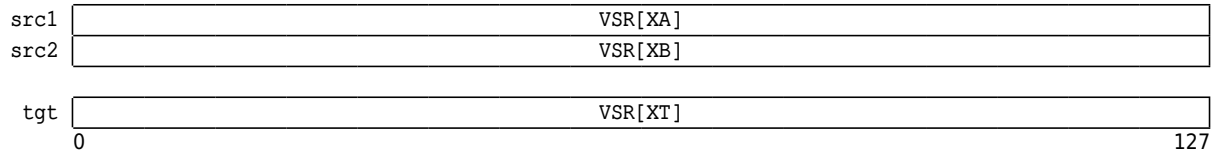
Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

The contents of VSR[XA] are exclusive-ORed with the contents of VSR[XB] and the result is placed into VSR[XT].

Special Registers Altered:

None

VSR Data Layout for xxlnor, xxlor, xxlorc, & xxlxor



7.6.2.13.2 VSX Vector Select Instruction

VSX Vector Select XX4-form

xxsel XT,XA,XB,XC

60	T	A	B	C	3	CX	AX	BX	TX
0	6	11	16	21	26	28	29	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

src1 ← VSR[32×AX+A]
src2 ← VSR[32×BX+B]
mask ← VSR[32×CX+C]

VSR[32×TX+T] ← (src1 & ¬mask) | (src2 & mask)
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.
Let XC be the value 32×CX + C.

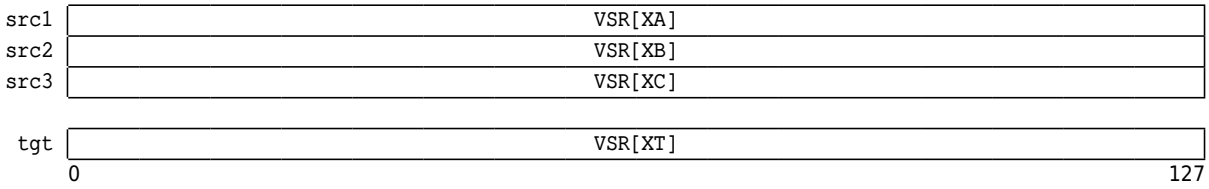
Let src1 be the contents of VSR[XA].
Let src2 be the contents of VSR[XB].
Let mask be the contents of VSR[XC].

The value, (src1 & ¬mask) | (src2 & mask), is placed into VSR[XT].

Special Registers Altered:

None

VSR Data Layout for xxsel



7.6.2.13.3 VSX Vector Evaluate Instruction

VSX Vector Evaluate 8RR:XX4-form

xxeval XT,XA,XB,XC,IMM

Prefix:

1	1	0	//	///	IMM
0	6	8	12	14	24 31

Suffix:

34	T	A	B	C	1	CX	AX	BX	TX
0	6	11	16	21	26	28	29	30	31

```
if MSR.VSX=0 then VSX_Unavailable()
src1 ← VSR[32xAX+A]
src2 ← VSR[32xBX+B]
src3 ← VSR[32xCX+C]
result ←
   (¬src1&¬src2&¬src3 & qword_bit_splat(IMM.bit[0])) |
   (¬src1&¬src2& src3 & qword_bit_splat(IMM.bit[1])) |
   (¬src1& src2&¬src3 & qword_bit_splat(IMM.bit[2])) |
   (¬src1& src2& src3 & qword_bit_splat(IMM.bit[3])) |
   ( src1&¬src2&¬src3 & qword_bit_splat(IMM.bit[4])) |
   ( src1&¬src2& src3 & qword_bit_splat(IMM.bit[5])) |
   ( src1& src2&¬src3 & qword_bit_splat(IMM.bit[6])) |
   ( src1& src2& src3 & qword_bit_splat(IMM.bit[7]))
VSR[32xTX+T] ← result
```

For each integer value `i`, 0 to 127, do the following.

Let `j` be the value of the concatenation of the contents of bit `i` of `VSR[XA]`, bit `i` of `VSR[XB]`, bit `i` of `VSR[XC]`.

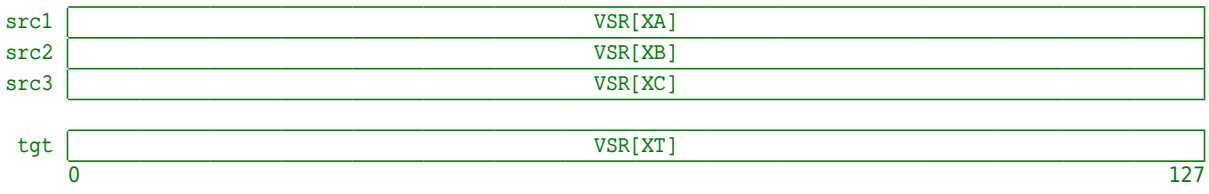
The value of bit `j` of `IMM` is placed into bit `i` of `VSR[XT]`.

See Table 7.92, "xxeval(A, B, C, IMM) Equivalent Functions," on page 911 for the equivalent function evaluated by this instruction for any given value of `IMM`.

Special Registers Altered:

None

VSR Data Layout for xxeval



IMM	0b.....00	0b.....01	0b.....10	0b.....11
0b000000..	false	and(A,B,C)	nor(C,nand(B,A))	and(B,A)
0b000001..	nor(B,nand(A,C))	and(C,A)	and(A,xor(B,C))	and(A,or(B,C))
0b000010..	and(A,nor(B,C))	and(A,eqv(B,C))	and(A,not(C))	C?and(B,A):A
0b000011..	and(A,not(B))	B?and(A,C):A	and(A,nand(B,C))	A
0b000100..	nor(A,nand(B,C))	and(C,B)	and(B,xor(A,C))	and(B,or(A,C))
0b000101..	and(C,xor(B,A))	and(C,or(A,B))	A?xor(B,C):and(B,C)	major(A,B,C)
0b000110..	A?nor(B,C):and(B,C)	A?eqv(B,C):and(B,C)	A?not(C):and(B,C)	C?B:A
0b000111..	A?not(B):and(B,C)	B?C:A	xor(A,and(B,C))	or(A,and(B,C))
0b001000..	and(B,nor(A,C))	and(B,eqv(A,C))	and(B,not(C))	C?and(B,A):B
0b001001..	B?nor(A,C):and(A,C)	B?eqv(A,C):and(A,C)	B?not(C):and(A,C)	C?A:B
0b001010..	nor(C,eqv(B,A))	C?and(B,A):xor(B,A)	nor(C,nor(B,A))	C?and(B,A):or(B,A)
0b001011..	B?nor(A,C):A	B?eqv(A,C):A	B?not(C):A	C?A:or(B,A)
0b001100..	and(B,not(A))	A?and(B,C):B	and(B,nand(A,C))	B
0b001101..	B?not(A):and(A,C)	A?C:B	xor(B,and(A,C))	or(B,and(A,C))
0b001110..	A?nor(B,C):B	A?eqv(B,C):B	A?not(C):B	C?B:or(B,A)
0b001111..	xor(B,A)	C?or(B,A):xor(B,A)	A?nand(B,C):B	or(B,A)
0b010000..	and(C,nor(B,A))	and(C,eqv(B,A))	C?nor(B,A):and(B,A)	C?eqv(B,A):and(B,A)
0b010001..	and(C,not(B))	B?and(A,C):C	C?not(B):and(B,A)	B?A:C
0b010010..	nor(B,eqv(A,C))	B?and(A,C):xor(A,C)	C?nor(B,A):A	B?A:xor(A,C)
0b010011..	nor(B,nor(A,C))	B?and(A,C):or(A,C)	C?not(B):A	B?A:or(A,C)
0b010100..	and(C,not(A))	A?and(B,C):C	C?not(A):and(B,A)	A?B:C
0b010101..	and(C,nand(B,A))	C	xor(C,and(B,A))	or(C,and(B,A))
0b010110..	A?nor(B,C):C	A?eqv(B,C):C	xor(C,A)	B?or(A,C):xor(A,C)
0b010111..	A?not(B):C	B?C:or(A,C)	A?nand(B,C):C	or(C,A)
0b011000..	nor(A,eqv(B,C))	A?and(B,C):xor(B,C)	C?nor(B,A):B	A?B:xor(B,C)
0b011001..	B?nor(A,C):C	A?C:xor(B,C)	xor(C,B)	A?or(B,C):xor(B,C)
0b011010..	A?nor(B,C):xor(B,C)	xor(A,B,C)	xor(C,or(B,A))	C?eqv(B,A):or(B,A)
0b011011..	xor(B,or(A,C))	B?eqv(A,C):or(A,C)	B?not(C):or(A,C)	or(A,xor(B,C))
0b011100..	nor(A,nor(B,C))	A?and(B,C):or(B,C)	C?not(A):B	A?B:or(B,C)
0b011101..	B?not(A):C	A?C:or(B,C)	B?nand(A,C):C	or(C,B)
0b011110..	xor(A,or(B,C))	A?eqv(B,C):or(B,C)	A?not(C):or(B,C)	or(B,xor(A,C))
0b011111..	A?not(B):or(B,C)	or(C,xor(B,A))	A?nand(B,C):or(B,C)	or(A,B,C)
0b100000..	nor(A,B,C)	A?and(B,C):nor(B,C)	nor(C,xor(B,A))	A?B:nor(B,C)
0b100001..	nor(B,xor(A,C))	A?C:nor(B,C)	A?xor(B,C):nor(B,C)	eqv(A,or(B,C))
0b100010..	nor(C,B)	B?and(A,C):not(C)	A?not(C):nor(B,C)	B?A:not(C)
0b100011..	A?not(B):nor(B,C)	C?A:not(B)	A?nand(B,C):nor(B,C)	or(A,nor(B,C))
0b100100..	nor(A,xor(B,C))	B?C:nor(A,C)	B?xor(A,C):nor(A,C)	eqv(B,or(A,C))
0b100101..	C?xor(B,A):nor(B,A)	eqv(C,or(B,A))	eqv(A,B,C)	A?or(B,C):eqv(B,C)
0b100110..	A?nor(B,C):eqv(B,C)	eqv(C,B)	A?not(C):eqv(B,C)	B?or(A,C):not(C)
0b100111..	A?not(B):eqv(B,C)	C?or(B,A):not(B)	A?nand(B,C):eqv(B,C)	or(A,eqv(B,C))
0b101000..	nor(C,A)	A?and(B,C):not(C)	B?not(C):nor(A,C)	A?B:not(C)
0b101001..	B?nor(A,C):eqv(A,C)	eqv(C,A)	A?xor(B,C):not(C)	A?or(B,C):not(C)
0b101010..	nor(C,and(B,A))	eqv(C,and(B,A))	not(C)	nand(C,nand(B,A))
0b101011..	A?not(B):not(C)	C?A:nand(B,A)	A?nand(B,C):not(C)	or(A,not(C))
0b101100..	B?not(A):nor(A,C)	C?B:not(A)	B?nand(A,C):nor(A,C)	or(B,nor(A,C))
0b101101..	B?not(A):eqv(A,C)	C?or(B,A):not(A)	B?nand(A,C):eqv(A,C)	or(B,eqv(A,C))
0b101110..	B?not(A):not(C)	C?B:nand(B,A)	B?nand(A,C):not(C)	or(B,not(C))
0b101111..	C?xor(B,A):nand(B,A)	C?or(B,A):nand(B,A)	nand(C,eqv(B,A))	nand(C,nor(B,A))
0b110000..	nor(B,A)	A?and(B,C):not(B)	C?nor(B,A):eqv(B,A)	eqv(B,A)
0b110001..	C?not(B):nor(B,A)	A?C:not(B)	A?xor(B,C):not(B)	A?or(B,C):not(B)
0b110010..	nor(B,and(A,C))	eqv(B,and(A,C))	A?not(C):not(B)	B?A:nand(A,C)
0b110011..	not(B)	nand(B,nand(A,C))	A?nand(B,C):not(B)	or(A,not(B))
0b110100..	C?not(A):nor(B,A)	B?C:not(A)	B?xor(A,C):not(A)	B?or(A,C):not(A)
0b110101..	C?nand(B,A):nor(B,A)	or(C,nor(B,A))	C?nand(B,A):eqv(B,A)	or(C,eqv(B,A))
0b110110..	C?not(A):not(B)	B?C:nand(A,C)	B?xor(A,C):nand(A,C)	B?or(A,C):nand(A,C)
0b110111..	C?nand(B,A):not(B)	or(C,not(B))	nand(B,eqv(A,C))	nand(B,nor(A,C))
0b111000..	nor(A,and(B,C))	eqv(A,and(B,C))	B?not(C):not(A)	A?B:nand(B,C)
0b111001..	C?not(B):not(A)	A?C:nand(B,C)	A?xor(B,C):nand(B,C)	A?or(B,C):nand(B,C)
0b111010..	minor(A,B,C)	A?eqv(B,C):nand(B,C)	nand(C,or(B,A))	nand(C,xor(B,A))
0b111011..	nand(B,or(A,C))	nand(B,xor(A,C))	nand(C,B)	or(A,nand(B,C))
0b111100..	not(A)	nand(A,nand(B,C))	B?nand(A,C):not(A)	or(B,not(A))
0b111101..	C?nand(B,A):not(A)	or(C,not(A))	nand(A,eqv(B,C))	nand(A,nor(B,C))
0b111110..	nand(A,or(B,C))	nand(A,xor(B,C))	nand(C,A)	or(B,nand(A,C))
0b111111..	nand(B,A)	or(C,nand(B,A))	nand(A,B,C)	true

Table 7.92: xxeval(A, B, C, IMM) Equivalent Functions

7.6.2.13.4 VSX Vector Blend Instructions

VSX Vector Blend Variable Byte 8RR:XX4-form

xxblendvb XT,XA,XB,XC

Prefix:

1	1	0	//	///
0	6	8	12	14 31

Suffix:

33	T	A	B	C	0	CX	AX	BX	TX
0	6	11	16	21	26	28	29	30	31

```
if MSR.VSX=0 then VSX_Unavailable()
do i = 0 to 15
   if VSR[32×CX+C].byte[i].bit[0]=0 then
      VSR[32×TX+T].byte[i] ← VSR[32*AX+A].byte[i]
   else
      VSR[32×TX+T].byte[i] ← VSR[32*BX+B].byte[i]
end
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.
Let XC be the value 32×CX + C.

For each integer value i from 0 to 15, do the following.

If the contents of bit 0 of byte element i of VSR[XC] is equal to 0, the contents of byte element i of VSR[XA] are placed into byte element i of VSR[XT]. Otherwise, the contents of byte element i of VSR[XB] are placed into byte element i of VSR[XT].

Special Registers Altered:

None

VSX Vector Blend Variable Doubleword 8RR:XX4-form

xxblendvd XT,XA,XB,XC

Prefix:

1	1	0	//	///
0	6	8	12	14 31

Suffix:

33	T	A	B	C	3	CX	AX	BX	TX
0	6	11	16	21	26	28	29	30	31

```
if MSR.VSX=0 then VSX_Unavailable()
do i = 0 to 1
   if VSR[32×CX+C].dword[i].bit[0]=0 then
      VSR[32×TX+T].dword[i] ← VSR[32×AX+A].dword[i]
   else
      VSR[32×TX+T].dword[i] ← VSR[32×BX+B].dword[i]
end
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.
Let XC be the value 32×CX + C.

For each integer value i from 0 to 1, do the following.

If the contents of bit 0 of doubleword element i of VSR[XC] is equal to 0, the contents of doubleword element i of VSR[XA] are placed into doubleword element i of VSR[XT]. Otherwise, the contents of doubleword element i of VSR[XB] are placed into doubleword element i of VSR[XT].

Special Registers Altered:

None

VSR Data Layout for xxblendvb

	.byte[0]	.byte[1]	.byte[2]	.byte[3]	.byte[4]	.byte[5]	.byte[6]	.byte[7]	.byte[8]	.byte[9]	.byte[10]	.byte[11]	.byte[12]	.byte[13]	.byte[14]	.byte[15]
src1																
src2																
src3																
tgt																

0 8 16 24 32 40 48 56 64 72 80 88 96 104 112 120 127

VSR Data Layout for xxblendvd

src1	VSR[XA].dword[0]	VSR[XA].dword[1]
src2	VSR[XB].dword[0]	VSR[XB].dword[1]
src3	VSR[XC].dword[0]	VSR[XC].dword[1]
tgt	VSR[XT].dword[0]	VSR[XT].dword[1]

0 64 127

VSX Vector Blend Variable Halfword 8RR:XX4-form

xxblendvh XT,XA,XB,XC

Prefix:

1	1	0	//	///
0	6	8	12	14 31

Suffix:

33	T	A	B	C	1	CX	AX	BX	TX
0	6	11	16	21	26	28	29	30	31

```
if MSR.VSX=0 then VSX_Unavailable()
do i = 0 to 7
   if VSR[32×CX+C].hword[i].bit[0]=0 then
      VSR[32×TX+T].hword[i] ← VSR[32×AX+A].hword[i]
   else
      VSR[32×TX+T].hword[i] ← VSR[32×BX+B].hword[i]
end
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.
Let XC be the value 32×CX + C.

For each integer value i from 0 to 7, do the following.

If the contents of bit 0 of halfword element i of VSR[XC] is equal to 0, the contents of halfword element i of VSR[XA] are placed into halfword element i of VSR[XT]. Otherwise, the contents of halfword element i of VSR[XB] are placed into halfword element i of VSR[XT].

Special Registers Altered:

None

VSX Vector Blend Variable Word 8RR:XX4-form

xxblendvw XT,XA,XB,XC

Prefix:

1	1	0	//	///
0	6	8	12	14 31

Suffix:

33	T	A	B	C	2	CX	AX	BX	TX
0	6	11	16	21	26	28	29	30	31

```
if MSR.VSX=0 then VSX_Unavailable()
do i = 0 to 3
   if VSR[32×CX+C].word[i].bit[0]=0 then
      VSR[32×TX+T].word[i] ← VSR[32×AX+A].word[i]
   else
      VSR[32×TX+T].word[i] ← VSR[32×BX+B].word[i]
end
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.
Let XC be the value 32×CX + C.

For each integer value i from 0 to 3, do the following.

If the contents of bit 0 of word element i of VSR[XC] is equal to 0, the contents of word element i of VSR[XA] are placed into word element i of VSR[XT]. Otherwise, the contents of word element i of VSR[XB] are placed into word element i of VSR[XT].

Special Registers Altered:

None

VSR Data Layout for xxblendvh

src1	VSR[XA].hword[0]	VSR[XA].hword[1]	VSR[XA].hword[2]	VSR[XA].hword[3]	VSR[XA].hword[4]	VSR[XA].hword[5]	VSR[XA].hword[6]	VSR[XA].hword[7]
src2	VSR[XB].hword[0]	VSR[XB].hword[1]	VSR[XB].hword[2]	VSR[XB].hword[3]	VSR[XB].hword[4]	VSR[XB].hword[5]	VSR[XB].hword[6]	VSR[XB].hword[7]
src3	VSR[XC].hword[0]	VSR[XC].hword[1]	VSR[XC].hword[2]	VSR[XC].hword[3]	VSR[XC].hword[4]	VSR[XC].hword[5]	VSR[XC].hword[6]	VSR[XC].hword[7]

tgt	VSR[XT].hword[0]	VSR[XT].hword[1]	VSR[XT].hword[2]	VSR[XT].hword[3]	VSR[XT].hword[4]	VSR[XT].hword[5]	VSR[XT].hword[6]	VSR[XT].hword[7]
	0	16	32	48	64	80	96	112 127

VSR Data Layout for xxblendvw

src1	VSR[XA].word[0]	VSR[XA].word[1]	VSR[XA].word[2]	VSR[XA].word[3]
src2	VSR[XB].word[0]	VSR[XB].word[1]	VSR[XB].word[2]	VSR[XB].word[3]
src3	VSR[XC].word[0]	VSR[XC].word[1]	VSR[XC].word[2]	VSR[XC].word[3]

tgt	VSR[XT].word[0]	VSR[XT].word[1]	VSR[XT].word[2]	VSR[XT].word[3]
	0	32	64	96 127

7.6.2.14 VSX Vector Permute-class Instructions

7.6.2.14.1 VSX Vector Byte-Reverse Instructions

VSX Vector Byte-Reverse Doubleword XX2-form

xxbrd XT,XB

60	T	23	B	475	BX	TX
0	6	11	16	21	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

do i = 0 to 1
vsrc ← VSR[32×BX+B].dword[i]
   do j = 0 to 7
      VSR[32×TX+T].dword[i].byte[j] ← vsrc.byte[7-j]
   end
end
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

For each integer value i from 0 to 1, do the following.

The contents of byte 7 of doubleword element i of VSR[XB] are placed into byte 0 of doubleword element i of VSR[XT].

The contents of byte 6 of doubleword element i of VSR[XB] are placed into byte 1 of doubleword element i of VSR[XT].

The contents of byte 5 of doubleword element i of VSR[XB] are placed into byte 2 of doubleword element i of VSR[XT].

The contents of byte 4 of doubleword element i of VSR[XB] are placed into byte 3 of doubleword element i of VSR[XT].

The contents of byte 3 of doubleword element i of VSR[XB] are placed into byte 4 of doubleword element i of VSR[XT].

The contents of byte 2 of doubleword element i of VSR[XB] are placed into byte 5 of doubleword element i of VSR[XT].

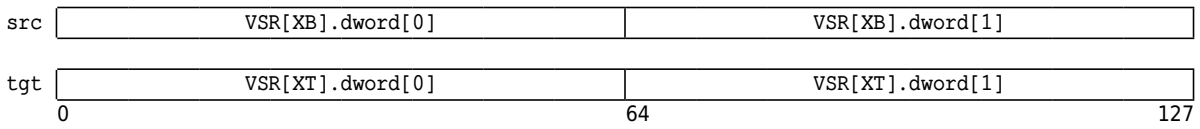
The contents of byte 1 of doubleword element i of VSR[XB] are placed into byte 6 of doubleword element i of VSR[XT].

The contents of byte 0 of doubleword element i of VSR[XB] are placed into byte 7 of doubleword element i of VSR[XT].

Special Registers Altered:

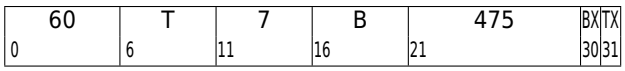
None

VSR Data Layout for xxbrd



VSX Vector Byte-Reverse Halfword XX2-form

xxbrh XT,XB




```
if MSR.VSX=0 then VSX_Unavailable()

do i = 0 to 7
   vsrc ← VSR[32×BX+B].hword[i]
   do j = 0 to 1
      VSR[32×TX+T].hword[i].byte[j] ← vsrc.byte[1-j]
   end
end
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

For each integer value i from 0 to 7, do the following.

The contents of byte 1 of halfword element i of VSR[XB] are placed into byte 0 of halfword element i of VSR[XT].

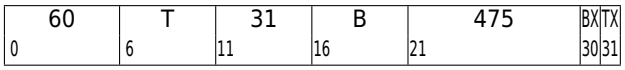
The contents of byte 0 of halfword element i of VSR[XB] are placed into byte 1 of halfword element i of VSR[XT].

Special Registers Altered:

None

VSX Vector Byte-Reverse Quadword XX2-form

xxbrq XT,XB




```
if MSR.VSX=0 then VSX_Unavailable()

do i = 0 to 15
   VSR[32×TX+T].byte[i] ← VSR[32×BX+B].byte[15-i]
end
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

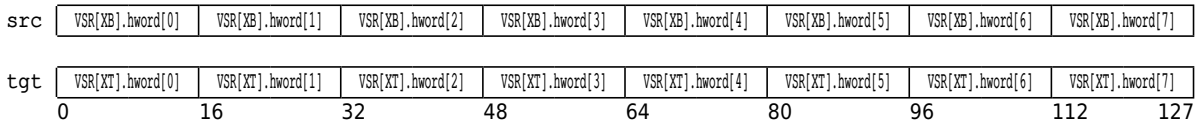
For each integer value i from 0 to 15, do the following.

The contents of byte sub-element 15-i of VSR[XB] are placed into byte sub-element i of VSR[XT].

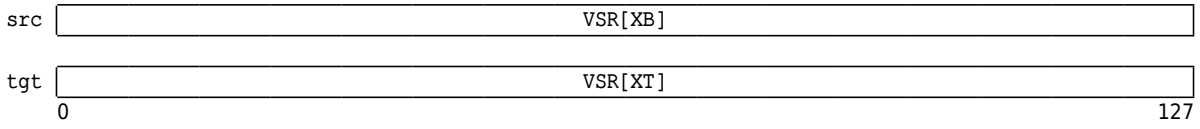
Special Registers Altered:

None

VSR Data Layout for xxbrh



VSR Data Layout for xxbrq



VSX Vector Byte-Reverse Word XX2-form

xxbrw XT,XB

60	T	15	B	475	BX	TX
0	6	11	16	21	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

do i = 0 to 3
   vsrc ← VSR[32×BX+B].word[i]
   do j = 0 to 3
      VSR[32×TX+T].word[i].byte[j] ← vsrc.byte[3-j]
   end
end
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

For each integer value i from 0 to 3, do the following.

The contents of byte 3 of word element i of VSR[XB] are placed into byte 0 of word element i of VSR[XT].

The contents of byte 2 of word element i of VSR[XB] are placed into byte 1 of word element i of VSR[XT].

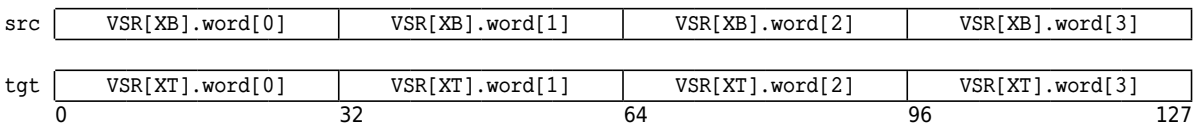
The contents of byte 1 of word element i of VSR[XB] are placed into byte 2 of word element i of VSR[XT].

The contents of byte 0 of word element i of VSR[XB] are placed into byte 3 of word element i of VSR[XT].

Special Registers Altered:

None

VSR Data Layout for xxbrw



7.6.2.14.2 VSX Vector Insert/Extract Instructions

VSX Vector Extract Unsigned Word XX2-form

xxextractuw XT,XB,UIM

60	T	/	UIM	B	165	BX	TX
0	6	11	12	16	21	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

src ← VSR[32×BX+B].byte[UIM:UIM+3]

VSR[32×TX+T].dword[0] ← EXTZ64(src)
VSR[32×TX+T].dword[1] ← 0x0000_0000_0000_0000
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

The contents of byte elements UIM:UIM+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 UIM is greater than 12, the results are undefined.

Special Registers Altered:

None

VSX Vector Insert Word XX2-form

xxinsertw XT,XB,UIM

60	T	/	UIM	B	181	BX	TX
0	6	11	12	16	21	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

VSR[32×TX+T].byte[UIM:UIM+3] ← VSR[32×BX+B].bit[32:63]
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

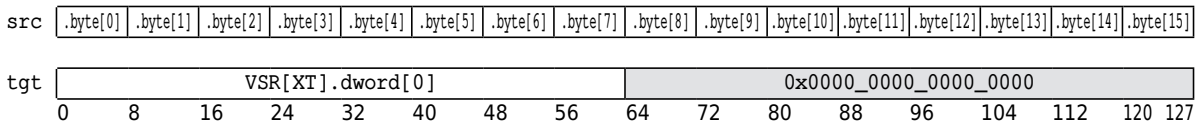
The contents of word element 1 of VSR[XB] are placed into byte elements UIM:UIM+3 of VSR[XT]. The contents of the remaining byte elements of VSR[XT] are not modified.

If the value of UIM is greater than 12, the results are undefined.

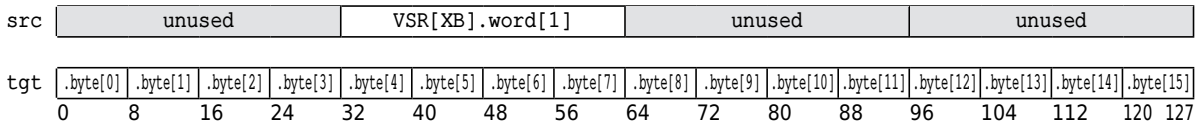
Special Registers Altered:

None

VSR Data Layout for xxextractuw



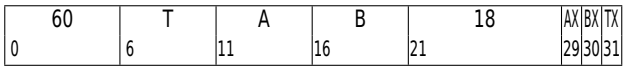
VSR Data Layout for xxinsertw



7.6.2.14.3 VSX Vector Merge Instructions

VSX Vector Merge High Word XX3-form

xxmrghw XT,XA,XB




```
if MSR.VSX=0 then VSX_Unavailable()

VSR[32×TX+T].word[0] ← VSR[32×AX+A].word[0]
VSR[32×TX+T].word[1] ← VSR[32×BX+B].word[0]
VSR[32×TX+T].word[2] ← VSR[32×AX+A].word[1]
VSR[32×TX+T].word[3] ← VSR[32×BX+B].word[1]
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

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].

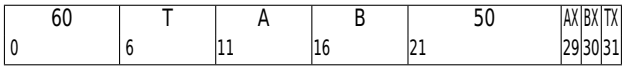
The contents of word element 1 of VSR[XB] are placed into word element 3 of VSR[XT].

Special Registers Altered:

None

VSX Vector Merge Low Word XX3-form

xxmrglw XT,XA,XB




```
if MSR.VSX=0 then VSX_Unavailable()

VSR[32×TX+T].word[0] ← VSR[32×AX+A].word[2]
VSR[32×TX+T].word[1] ← VSR[32×BX+B].word[2]
VSR[32×TX+T].word[2] ← VSR[32×AX+A].word[3]
VSR[32×TX+T].word[3] ← VSR[32×BX+B].word[3]
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

The contents of word element 2 of VSR[XA] are placed into word element 0 of VSR[XT].

The contents of word element 2 of VSR[XB] are placed into word element 1 of VSR[XT].

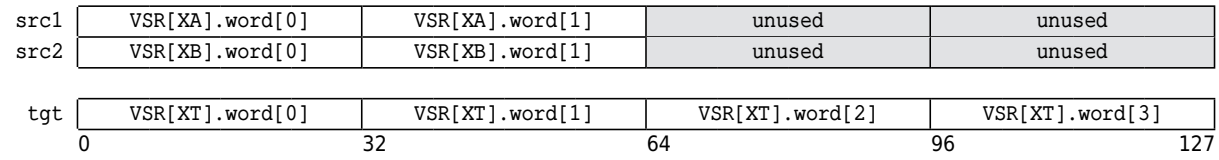
The contents of word element 3 of VSR[XA] are placed into word element 2 of VSR[XT].

The contents of word element 3 of VSR[XB] are placed into word element 3 of VSR[XT].

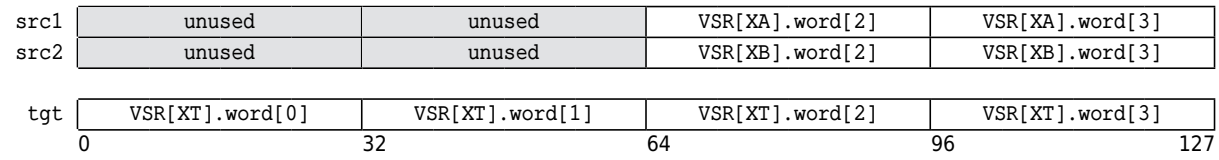
Special Registers Altered:

None

VSR Data Layout for xxmrghw



VSR Data Layout for xxmrglw



7.6.2.14.4 VSX Vector Splat Instructions

VSX Vector Splat Immediate32 Doubleword Indexed 8RR:D-form

xxsplti32dx XT,IX,IMM32

Prefix:

1	1	0	//	//	imm0
0	6	8	12	14	16 31

Suffix:

32	T	0	IX	TX	imm1
0	6	11	14	15	16 31

```
if MSR.VSX=0 then VSX_Unavailable()
IMM32 ← imm0<<16 | imm1

VSR[32×TX+T].dword[0].word[IX] ← IMM32
VSR[32×TX+T].dword[1].word[IX] ← IMM32
```

Let XT be the value 32×TX + T.
Let IMM32 be the concatenation of imm0 and imm1.

IMM32 is placed into word element IX of each doubleword element of VSR[XT]. The contents of the remaining word elements are not modified.

Special Registers Altered:

None

VSX Vector Splat Immediate Byte X-form

xxspltib XT,IMM8

60	T	0	IMM8	360	TX
0	6	11	13	21	31

```
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] ← UIM8
end
```

Let XT be the sum 32×TX + T.

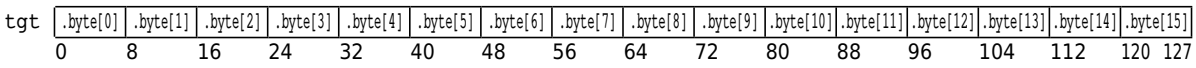
The value IMM8 is copied into each byte element of VSR[XT].

Special Registers Altered:

None

VSR Data Layout for xxsplti32dx

VSR Data Layout for xxspltib



VSX Vector Splat Immediate Double-Precision 8RR:D-form

xxspltidp XT,IMM32

Prefix:

1	1	0	//	//	imm0
0	6	8	12	14	16 31

Suffix:

32	T	2	TX	imm1
0	6	11	15	16 31

```
if MSR.VSX=0 then VSX_Unavailable()

IMM32 ← imm0<<16 | imm1;

VSR[32×TX+T].dword[0] ← bfp64_CONVERT_FROM_BFP(IMM32);
VSR[32×TX+T].dword[1] ← bfp64_CONVERT_FROM_BFP(IMM32);
```

Let IMM32 be the concatenation of imm0 and imm1, representing a single-precision value.

IMM32 is converted to double-precision format and placed into each doubleword element of VSR[XT].

If IMM32 specifies a single-precision denormal value (i.e., bits 1:8 equal to 0 and bits 9:31 not equal to 0), the result is undefined.

Special Registers Altered:

None

VSX Vector Splat Immediate Word 8RR:D-form

xxspltiw XT,IMM32

Prefix:

1	1	0	//	//	imm0
0	6	8	12	14	16 31

Suffix:

32	T	3	TX	imm1
0	6	11	15	16 31

```
if MSR.VSX=0 then VSX_Unavailable()
IMM32 ← imm0<<16 | imm1

do i = 0 to 3
   VSR[32×TX+T].word[i] ← IMM32
end
```

Let XT be the value 32×TX + T.
Let IMM32 be the concatenation of imm0 and imm1.

IMM32 is placed into each word element of VSR[XT].

Special Registers Altered:

None

VSR Data Layout for xxspltidp

tgt	VSR[XT].dword[0]	VSR[XT].dword[1]
	0 64	127

VSR Data Layout for xxspltiw

tgt	VSR[XT].word[0]	VSR[XT].word[1]	VSR[XT].word[2]	VSR[XT].word[3]
	0 32	64	96	127

VSX Vector Splat Word XX2-form

xxspltw XT,XB,UIM

60	T	///	UIM	B	164	BX	TX
0	6	11	14	16	21	30	31

```
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]
VSR[32×TX+T].word[2] ← VSR[32×BX+B].word[UIM]
VSR[32×TX+T].word[3] ← VSR[32×BX+B].word[UIM]
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

The contents of word element UIM of VSR[XB] are replicated in each word element of VSR[XT].

Special Registers Altered:

None

VSR Data Layout for xxspltw

src	VSR[XB].word[0]	VSR[XB].word[1]	VSR[XB].word[2]	VSR[XB].word[3]
tgt	VSR[XT].word[0]	VSR[XT].word[1]	VSR[XT].word[2]	VSR[XT].word[3]
	0	32	64	96 127

7.6.2.14.5 VSX Vector Permute Instructions

VSX Vector Permute Doubleword Immediate XX3-form

xxpermdi XT,XA,XB,DM

60	T	A	B	0	DM	10	AX	BX	TX
0	6	11	16	21	22	24	29	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

VSR[32×TX+T].dword[0] ← VSR[32×AX+A].dword[DM.bit[0]]
VSR[32×TX+T].dword[1] ← VSR[32×BX+B].dword[DM.bit[1]]
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

If DM.bit[0]=0, the contents of doubleword element 0 of VSR[XA] are placed into doubleword element 0 of VSR[XT]. Otherwise the contents of doubleword element 1 of VSR[XA] are placed into doubleword element 0 of VSR[XT].

If DM.bit[1]=0, the contents of doubleword element 0 of VSR[XB] are placed into doubleword element 1 of VSR[XT]. Otherwise the contents of doubleword element 1 of VSR[XB] are placed into doubleword element 1 of VSR[XT].

Special Registers Altered:

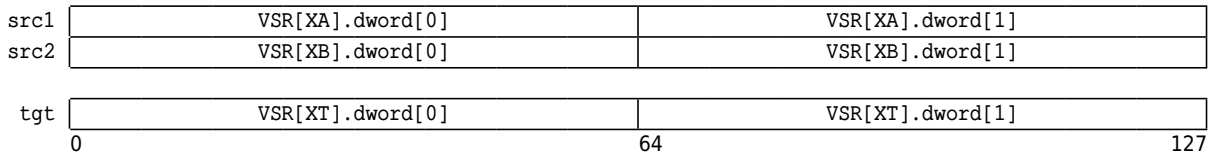
None

Extended Mnemonics:

Extended Mnemonics for *VSX Vector Permute Doubleword Immediate*:

Extended mnemonic:		Equivalent to:
xxspltd	XT,XA,0	xxpermdi XT,XA,XA,0b00
xxspltd	XT,XA,1	xxpermdi XT,XA,XA,0b11
xxmrghd	XT,XA,XB	xxpermdi XT,XA,XB,0b00
xxmrgld	XT,XA,XB	xxpermdi XT,XA,XB,0b11
xxswapd	XT,XA	xxpermdi XT,XA,XA,0b10

VSR Data Layout for xxpermdi



VSX Vector Permute XX3-form

xxperm XT,XA,XB

60	T	A	B	26	AX	BX	TX
0	6	11	16	21	29	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

src.byte[0:15]  ← VSR[32×AX+A]
src.byte[16:31] ← VSR[32×TX+T]
pcv.byte[0:15]  ← VSR[32×BX+B]

do i = 0 to 15
   idx ← pcv.byte[i].bit[3:7]
   VSR[32×TX+T].byte[i] ← src.byte[idx]
end
```

Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.
Let XT be the value 32×TX + T.

Let bytes 0:15 of src be the contents of VSR[XA].
Let bytes 16:31 of src be the contents of VSR[XT].

Let the permute control vector pcv be the contents of VSR[XB].

For each integer value i from 0 to 15, do the following.

Let idx be the unsigned integer in bits 3:7 of byte element i of pcv.

The contents of byte element idx of src is placed into byte element i of VSR[XT].

Special Registers Altered:

None

VSX Vector Permute Right-indexed XX3-form

xxpermr XT,XA,XB

60	T	A	B	58	AX	BX	TX
0	6	11	16	21	29	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

src.byte[0:15]  ← VSR[32×AX+A]
src.byte[16:31] ← VSR[32×TX+T]
pcv.byte[0:15]  ← VSR[32×BX+B]

do i = 0 to 15
   idx ← pcv.byte[i].bit[3:7]
   VSR[32×TX+T].byte[i] ← src.byte[31-idx]
end
```

Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.
Let XT be the value 32×TX + T.

Let bytes 0:15 of src be the contents of VSR[XA].
Let bytes 16:31 of src be the contents of VSR[XT].

Let the permute control vector pcv be the contents of VSR[XB].

For each integer value i from 0 to 15, do the following.

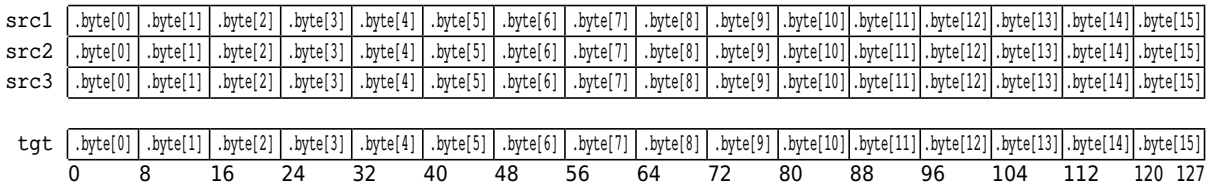
Let idx be the unsigned integer in bits 3:7 of byte element i of pcv.

The contents of byte element 31-idx of src is placed into byte element i of VSR[XT].

Special Registers Altered:

None

VSR Data Layout for xxperm & xxpermr



VSX Vector Permute Extended 8RR:XX4-form

xxpermx XT,XA,XB,XC,UIM

Prefix:

1	1	0	//	///	UIM
0	6	8	12	14	29 31

Suffix:

34	T	A	B	C	0	CX	AX	BX	TX
0	6	11	16	21	26	28	29	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

src.qword[0] ← VSR[32×AX+A]
src.qword[1] ← VSR[32×BX+B]
do i = 0 to 15
   section ← VSR[32×CX+C].byte[i].bit[0:2]
   eidx    ← VSR[32×CX+C].byte[i].bit[3:7]
   if section=UIM then
      VSR[32×TX+T].byte[i] ← src.byte[eidx]
   else
      VSR[32×TX+T].byte[i] ← 0x00
end
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.
Let XC be the value 32×CX + C.

Let UIM specify which 32-byte section of the long vector that src contains.

Let src be the concatenation VSR[XA] and VSR[XB], comprising a 32-byte section of up to a 256-byte vector.

For each integer value i from 0 to 15, do the following.

Let eidx be the contents of bits 3:7 of byte element i of VSR[XC].

If UIM is equal to the contents of bits 0:2 of byte element i of VSR[XC], the contents of byte element eidx of src are placed into byte element i of VSR[XT]. Otherwise, the contents of byte element i of VSR[XT] are set to 0.

Special Registers Altered:

None

Programming Note

The following is an example of emulating 256-bit ***xxperm***, where a 256-bit vector is contained in a pair of VSRs. The instruction is capable of emulating up to a 2048-bit *xxperm*.

```
vT0a = xxpermx(vA0, vA1, vC0, 0);
vT0b = xxpermx(vB0, vB1, vC0. 1);
vT1a = xxpermx(vA0, vA1, vC1, 0);
vT1b = xxpermx(vB0, vB1, vC1, 1);
vT0  = xxlor(vT0a, vT0b);
vT1  = xxlor(vT1a, vT1b);
```

Programming Note

The following is an example of a parallel table lookup. In this case, a 16-way SIMD 256-entry byte table lookup.

```
vT0 = xxpermx(vS0, vS1, vINDEX, 0);
vT1 = xxpermx(vS2, vS3, vINDEX, 1);
vT2 = xxpermx(vS4, vS5, vINDEX, 2);
vT3 = xxpermx(vS6, vS7, vINDEX, 3);
vT4 = xxpermx(vS8, vS9, vINDEX, 4);
vT5 = xxpermx(vSA, vSB, vINDEX, 5);
vT6 = xxpermx(vSC, vSD, vINDEX, 6);
vT7 = xxpermx(vSE, vSF, vINDEX, 7);

vT0  = xxlor(vT0, vT1);
vT1  = xxlor(vT2, vT3);
vT2  = xxlor(vT4, vT5);
vT3  = xxlor(vT6, vT7);

vT0  = xxlor(vT0, vT1);
vT1  = xxlor(vT2, vT3);
```

VSR Data Layout for xxpermx

	.byte[0]	.byte[1]	.byte[2]	.byte[3]	.byte[4]	.byte[5]	.byte[6]	.byte[7]	.byte[8]	.byte[9]	.byte[10]	.byte[11]	.byte[12]	.byte[13]	.byte[14]	.byte[15]
src1																
src2																
src3																
tgt																
	0	8	16	24	32	40	48	56	64	72	80	88	96	104	112	120 127

7.6.2.14.6 VSX Vector Shift Left Double Instructions

VSX Vector Shift Left Double by Word Immediate XX3-form

xxsldwi XT,XA,XB,SHW

60	T	A	B	0	SHW	2	AX	BX	TX
0	6	11	16	21	22	24	29	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

source.qword[0] ← VSR[32×AX+A]
source.qword[1] ← VSR[32×BX+B]
VSR[32×TX+T] ← source.word[SHW:SHW+3]
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

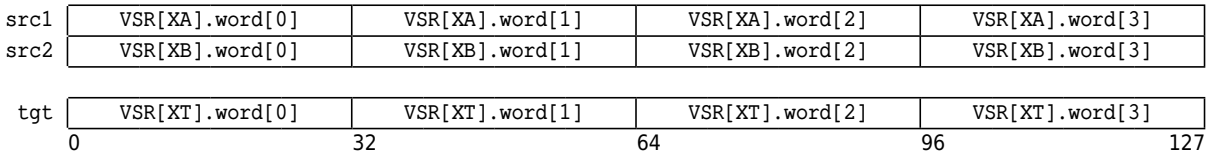
Let vsrc be the concatenation of the contents of VSR[XA] followed by the contents of VSR[XB].

Words SHW:SHW+3 of vsrc are placed into VSR[XT].

Special Registers Altered:

None

VSR Data Layout for xxsldwi



7.6.2.14.7 VSX Vector Generate Permute Control Vector Instructions

VSX Vector Generate PCV from Byte Mask X-form

xxgenpcvbm XT,VRB,IMM

60	T	IMM	VRB	916	TX
0	6	11	16	21	31

```
if MSR.VSX=0 then VSX_Unavailable()

XT ← 32×TX+T
if IMM=0b00000 then do  // Big-Endian expansion
   j ← 0
   do i = 0 to 15
      if VSR[VRB+32].byte[i].bit[0]=1 then do
         VSR[XT].byte[i] ← j
         j ← j + 1
      end
      else
         VSR[XT].byte[i] ← i + 0x10
   end
end

else if IMM=0b00001 then do  // Big-Endian compression
   j ← 0
   do i = 0 to 15
      if VSR[VRB+32].byte[i].bit[0]=1 then do
         VSR[XT].byte[j] = i
         j = j + 1
      end
   end
   do i = j to 15
      VSR[XT].byte[i] = 0xUU
   end
end

else if IMM=0b00010 then do  // Little-Endian expansion
   j ← 0;
   do i = 0 to 15
      if VSR[VRB+32].byte[15-i].bit[0]=1 then do
         VSR[XT].byte[15-i] ← j
         j ← j + 1
      end
      else
         VSR[XT].byte[15-i] ← i + 0x10
   end
end

else if IMM=0b00011 then do  // Little-Endian compression
   j ← 0
   do i = 0 to 15
      if VSR[VRB+32].byte[15-i].bit[0]=1 then do
         VSR[XT].byte[15-j] ← i
         j ← j + 1
      end
   end
   do i = j to 15
         VSR[XT].byte[15-i] ← 0xUU
   end
end
```

Let XT be the value 32×TX + T.

If IMM=0b00000, let pcv be the permute control vector required to enable a left-indexed permute (*vperm* or ***xxperm***) to implement an expansion of the leftmost byte elements of a source vector into the byte elements of a result vector specified by the byte-element mask in VSR[VRB+32].

If IMM=0b00001, let pcv be the permute control vector required to enable a left-indexed permute (*vperm* or ***xxperm***) to implement a compression of the sparse byte elements in a source vector specified by the byte-element mask in VSR[VRB+32] into the leftmost byte elements of a result vector.

If IMM=0b00010, let pcv be the permute control vector required to enable a right-indexed permute (*vpermr* or ***xxpermr***) to implement an expansion of the rightmost byte elements of a source vector into the byte elements of a result vector specified by the byte-element mask in VSR[VRB+32].

If IMM=0b00011, let pcv be the permute control vector required to enable a right-indexed permute (*vpermr* or ***xxpermr***) to implement a compression of the sparse byte elements in a source vector specified by the byte-element mask in VSR[VRB+32] into the rightmost byte elements of a result vector.

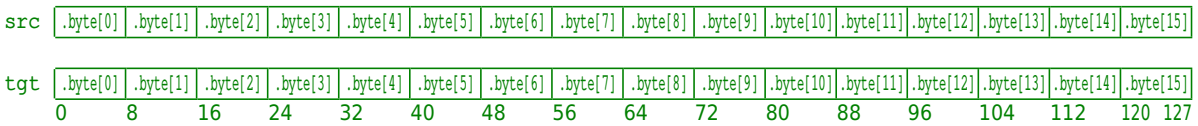
pcv is placed into VSR[XT].

Unused values of IMM are reserved.

Special Registers Altered:

None

VSR Data Layout for xxgenpcvbm



Programming Note

The following is an example of how a *Load VSX Vector and Expand Byte*, when using Big-Endian byte-ordering, can be emulated using *VSX Vector Generate PCV from Mask Byte*.

```
xxgenpcvbm  vPCV, vMASK, 0b00000        // generates the required permute control vector
                                        // for Big-Endian expansion
vcntmbb     vN, vMASK, 0b1              // calculates N, number of true byte-mask elements
lxvl        vLD, EA, rN                 // loads N bytes

// Option 1: expand & merge
xxperm      vT, vLD, vT, vPCV           // perform the expansion.
                                        // specifying vT as 2nd source operand causes expanded
                                        // load data to be merged into VSR[vT]
// Option 2: expand & zero
xxperm      vT, vLD, vZERO, vPCV        // perform the expansion,
                                        // specifying vZERO (vector of 0s) as 2nd source operand
                                        // causes expanded load data to be placed into VSR[vT]
                                        // with other elements set to 0
```

The following is an example of how a *Load VSX Vector Expand Byte*, when using Little-Endian byte-ordering, can be emulated using *VSX Vector Generate PCV from Mask Byte*.

```
xxgenpcvbm  vPCV, vMASK, 0b00010        // generates the required permute control vector
                                        // for Big-Endian expansion
vcntmbb     vN, vMASK, 0b1              // calculates N, number of true byte-mask elements
lxvl        vLD, EA, rN                 // loads N bytes

// Option 1: expand & merge
xxpermr     vT, vLD, vT, vPCV           // perform the expansion,
                                        // specifying vT as 2nd source operand causes expanded
                                        // load data to be merged into VSR[vT]
// Option 2: expand & zero
xxpermr     vT, vLD, vZERO, vPCV        // perform the expansion,
                                        // specifying vZERO (vector of 0s) as 2nd source operand
                                        // causes expanded load data to be placed into VSR[vT]
                                        // with other elements set to 0
```

The following is an example of how a *VSX Vector Compress Byte and Store*, when using Big-Endian byte-ordering, can be emulated using *VSX Vector Generate PCV from Mask Byte*.

```
xxgenpcvbm  vPCV, vMASK, 0b00001        // generates the required permute control vector
                                        // for Big-Endian compression
vcntmbb     vN, vMASK, 0b1              // calculates N, number of true byte-mask elements
xxperm      vSD, vS, vS, vPCV           // perform the compression
stxvl       vSD, rEA, rN                // store N bytes
```

The following is an example of how a *VSX Vector Byte Compress and Store*, when using Little-Endian byte-ordering, can be emulated using *VSX Vector Generate PCV from Mask*.

```
xxgenpcvbm  vPCV, vMASK, 0b00011        // generates the required permute control vector
                                        // for Big-Endian compression
vcntmbb     vN, vMASK, 0b1              // calculates N, number of true byte-mask elements
xxpermr     vSD, vS, vS, vPCV           // perform the compression
stxvl       vSD, rEA, rN                // store N bytes
```

VSX Vector Generate PCV from Doubleword Mask X-form

xxgenpcvdm XT,VRB,IMM

60	T	IMM	VRB	949	TX
0	6	11	16	21	31

```
if MSR.VSX=0 then VSX_Unavailable()

XT ← 32×TX+T
if IMM=0b00000 then do  // Big-Endian expansion
   j ← 0
   do i = 0 to 1
      if VSR[VRB+32].dword[i].bit[0]=1 then do
         VSR[XT].dword[i].byte[0] ← 8×j + 0x00
         VSR[XT].dword[i].byte[1] ← 8×j + 0x01
         VSR[XT].dword[i].byte[2] ← 8×j + 0x02
         VSR[XT].dword[i].byte[3] ← 8×j + 0x03
         VSR[XT].dword[i].byte[4] ← 8×j + 0x04
         VSR[XT].dword[i].byte[5] ← 8×j + 0x05
         VSR[XT].dword[i].byte[6] ← 8×j + 0x06
         VSR[XT].dword[i].byte[7] ← 8×j + 0x07
         j ← j + 1
      end
      else do
         VSR[XT].dword[i].byte[0] ← 8×i + 0x10
         VSR[XT].dword[i].byte[1] ← 8×i + 0x11
         VSR[XT].dword[i].byte[2] ← 8×i + 0x12
         VSR[XT].dword[i].byte[3] ← 8×i + 0x13
         VSR[XT].dword[i].byte[4] ← 8×i + 0x14
         VSR[XT].dword[i].byte[5] ← 8×i + 0x15
         VSR[XT].dword[i].byte[6] ← 8×i + 0x16
         VSR[XT].dword[i].byte[7] ← 8×i + 0x17
      end
   end
end

else if IMM=0b00001 then do  // Big-Endian compression
   j ← 0
   do i = 0 to 1
      if VSR[VRB+32].dword[i].bit[0]=1 then do
         VSR[XT].dword[j].byte[0] ← 8×i + 0x00
         VSR[XT].dword[j].byte[1] ← 8×i + 0x01
         VSR[XT].dword[j].byte[2] ← 8×i + 0x02
         VSR[XT].dword[j].byte[3] ← 8×i + 0x03
         VSR[XT].dword[j].byte[4] ← 8×i + 0x04
         VSR[XT].dword[j].byte[5] ← 8×i + 0x05
         VSR[XT].dword[j].byte[6] ← 8×i + 0x06
         VSR[XT].dword[j].byte[7] ← 8×i + 0x07
         j ← j + 1
      end
   end
   do i = j to 1
      VSR[XT].dword[i] ← 0xUUUU_UUUU_UUUU_UUUU
   end
end
else if IMM=0b00010 then do  // Little-Endian expansion
   j ← 0
   do i = 0 to 1
      if VSR[VRB+32].dword[1-i].bit[0]=1 then do
         VSR[XT].dword[1-i].byte[7] ← 8×j + 0x00
         VSR[XT].dword[1-i].byte[6] ← 8×j + 0x01
         VSR[XT].dword[1-i].byte[5] ← 8×j + 0x02
         VSR[XT].dword[1-i].byte[4] ← 8×j + 0x03
         VSR[XT].dword[1-i].byte[3] ← 8×j + 0x04
         VSR[XT].dword[1-i].byte[2] ← 8×j + 0x05
         VSR[XT].dword[1-i].byte[1] ← 8×j + 0x06
         VSR[XT].dword[1-i].byte[0] ← 8×j + 0x07
         j ← j + 1
      end
      else do
         VSR[XT].dword[1-i].byte[7] ← 8×i + 0x10
         VSR[XT].dword[1-i].byte[6] ← 8×i + 0x11
         VSR[XT].dword[1-i].byte[5] ← 8×i + 0x12
         VSR[XT].dword[1-i].byte[4] ← 8×i + 0x13
         VSR[XT].dword[1-i].byte[3] ← 8×i + 0x14
         VSR[XT].dword[1-i].byte[2] ← 8×i + 0x15
         VSR[XT].dword[1-i].byte[1] ← 8×i + 0x16
         VSR[XT].dword[1-i].byte[0] ← 8×i + 0x17
      end
   end
end

else if IMM=0b00011 then do  // Little-Endian compression
   j ← 0
   do i = 0 to 1
      if VSR[VRB+32].dword[1-i].bit[0]=1 then do
         VSR[XT].dword[1-j].byte[7] ← 8×i + 0x00
         VSR[XT].dword[1-j].byte[6] ← 8×i + 0x01
         VSR[XT].dword[1-j].byte[5] ← 8×i + 0x02
         VSR[XT].dword[1-j].byte[4] ← 8×i + 0x03
         VSR[XT].dword[1-j].byte[3] ← 8×i + 0x04
         VSR[XT].dword[1-j].byte[2] ← 8×i + 0x05
         VSR[XT].dword[1-j].byte[1] ← 8×i + 0x06
         VSR[XT].dword[1-j].byte[0] ← 8×i + 0x07
         j ← j + 1
      end
   end
   do i = j to 1
         VSR[XT].dword[1-i] ← 0xUUUU_UUUU_UUUU_UUUU
   end
end
```

Let XT be the value 32×TX + T.

If IMM=0b00000, let pcv be the permute control vector required to enable a left-indexed permute (*vperm* or ***xxperm***) to implement an expansion of the leftmost doubleword elements of a source vector into the doubleword elements of a result vector specified by the doubleword-element mask in VSR[VRB+32].

If IMM=0b00001, let pcv be the the permute control vector required to enable a left-indexed permute (*vperm* or ***xxperm***) to implement a compression of the sparse doubleword elements in a source vector specified by the doubleword-element mask in VSR[VRB+32] into the leftmost doubleword elements of a result vector.

If IMM=0b00010, let pcv be the permute control vector required to enable a right-indexed permute (*vpermr* or ***xxpermr***) to implement an expansion of the rightmost doubleword elements of a source vector into the doubleword elements of a result vector specified by the doubleword-element mask in VSR[VRB+32].

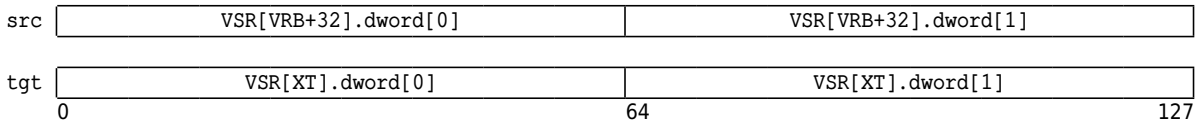
If IMM=0b00011, let pcv be the permute control vector required to enable a right-indexed permute (*vpermr* or ***xxpermr***) to implement a compression of the sparse doubleword elements in a source vector specified by the doubleword-element mask in VSR[VRB+32] into the rightmost doubleword elements of a result vector.

pcv is placed into VSR[XT].

Special Registers Altered:

None

VSR Data Layout for xxgenpcvdm



Programming Note

The following is an example of how a *Load VSX Vector Expand Doubleword*, when using Big-Endian byte-ordering, can be emulated using *VSX Vector Generate PCV from Mask Doubleword*.

```
xxgenpcvdm  vPCV, vMASK, 0b00000        // generates the required permute control vector
                                        // for Big-Endian expansion
vcntmbd     rN, vMASK, 0b1              // calculates N, number of true doubleword-mask elements,
                                        // adjusted to # of bytes
lxvl        vLD, EA, rN                 // loads 8×N bytes

// Option 1: expand & merge
xxperm      vT, vLD, vT, vPCV           // perform the expansion,
                                        // specifying vT as 2nd source operand causes expanded
                                        // load data to be merged into VSR[vT]
// Option 2: expand & zero
xxperm      vT, vLD, vZERO, vPCV        // perform the expansion,
                                        // specifying vZERO (vector of 0s) as 2nd source operand
                                        // causes expanded load data to be placed into VSR[vT]
                                        // with other elements set to 0
```

The following is an example of how a *Load VSX Vector Expand Doubleword*, when using Little-Endian byte-ordering, can be emulated using *VSX Vector Generate PCV from Mask Doubleword*.

```
xxgenpcvdm  vPCV, vMASK, 0b00010        // generates the required permute control vector
                                        // for Little-Endian expansion
vcntmbd     rN, vMASK, 0b1              // calculates N, number of true doubleword-mask elements,
                                        // adjusted to # of bytes
lxvl        vLD, EA, rN                 // loads 8×N bytes

// Option 1: expand & merge
xxpermr     vT, vLD, vT, vPCV           // perform the expansion,
                                        // specifying vT as 2nd source operand causes expanded
                                        // load data to be merged into VSR[vT]
// Option 2: expand & zero
xxpermr     vT, vLD, vZERO, vPCV        // perform the expansion,
                                        // specifying vZERO (vector of 0s) as 2nd source operand
                                        // causes expanded load data to be placed into VSR[vT]
                                        // with other elements set to 0
```

The following is an example of how a *Store VSX Vector Compress Doubleword*, when using Big-Endian byte-ordering, can be emulated using *VSX Vector Generate PCV from Mask Doubleword*.

```
xxgenpcvdm  vPCV, vMASK, 0b00001        // generates the required permute control vector
                                        // for Big-Endian compression
vcntmbd     rN, vMASK, 0b1              // calculates N, number of true doubleword-mask elements,
                                        // adjusted to # of bytes
xxperm      vSD, vS, vS, vPCV           // perform the compression
stxvl       vSD, rEA, rN                // store 8×N bytes
```

The following is an example of how a *Store VSX Vector Compress Doubleword*, when using Little-Endian byte-ordering, can be emulated using *VSX Vector Generate PCV from Mask Doubleword*.

```
xxgenpcvdm  vPCV, vMASK, 0b00011        // generates the required permute control vector
                                        // for Little-Endian compression
vcntmbd     rN, vMASK, 0b1              // calculates N, number of true doubleword-mask elements,
                                        // adjusted to # of bytes
xxpermr     vSD, vS, vS, vPCV           // perform the compression
stxvl       vSD, rEA, rN                // store 8×N bytes
```

VSX Vector Generate PCV from Halfword Mask X-form

xxgenpcvhm XT,VRB,IMM

60	T	IMM	VRB	917	TX
0	6	11	16	21	31

```
if MSR.VSX=0 then VSX_Unavailable()

XT ← 32×TX+T
if IMM=0b00000 then do  // Big-Endian expansion
   j ← 0
   do i = 0 to 7
      if VSR[VRB+32].hword[i].bit[0]=1 then do
         VSR[XT].hword[i].byte[0] ← 2×j + 0x00
         VSR[XT].hword[i].byte[1] ← 2×j + 0x01
         j ← j + 1
      end
      else do
         VSR[XT].hword[i].byte[0] ← 2×i + 0x10
         VSR[XT].hword[i].byte[1] ← 2×i + 0x11
      end
   end
end

else if IMM=0b00001 then do  // Big-Endian compression
   j ← 0
   do i = 0 to 7
      if VSR[VRB+32].hword[i].bit[0]=1 then do
         VSR[XT].hword[j].byte[0] ← 2×i + 0x00
         VSR[XT].hword[j].byte[1] ← 2×i + 0x01
         j ← j + 1
      end
   end
   do i = j to 7
      VSR[XT].hword[i] ← 0xUUUU
   end
end

else if IMM=0b00010 then do  // Little-Endian expansion
   j ← 0
   do i = 0 to 7
      if VSR[VRB+32].hword[7-i].bit[0]=1 then do
         VSR[XT].hword[7-i].byte[1] ← 2×j + 0x00
         VSR[XT].hword[7-i].byte[0] ← 2×j + 0x01
         j ← j + 1
      end
      else do
         VSR[XT].hword[7-i].byte[1] ← 2×i + 0x10
         VSR[XT].hword[7-i].byte[0] ← 2×i + 0x11
      end
   end
end

else if IMM=0b00011 then do  // Little-Endian compression
   j ← 0
   do i = 0 to 7
      if VSR[VRB+32].hword[7-i].bit[0]=1 then do
         VSR[XT].hword[7-j].byte[1] ← 2×i + 0x00
         VSR[XT].hword[7-j].byte[0] ← 2×i + 0x01
         j ← j + 1
      end
   end
   do i = j to 7
         VSR[XT].hword[7-i] ← 0xUUUU
   end
end
```

Let XT be the value 32×TX + T.

If IMM=0b00000, let pcv be the permute control vector required to enable a left-indexed permute (*vperm* or ***xxperm***) to implement an expansion of the leftmost halfword elements of a source vector into the halfword elements of a result vector specified by the halfword-element mask in VSR[VRB+32].

If IMM=0b00001, let pcv be the permute control vector required to enable a left-indexed permute (*vperm* or ***xxperm***) to implement a compression of the sparse halfword elements in a source vector specified by the halfword-element mask in VSR[VRB+32] into the leftmost halfword elements of a result vector.

If IMM=0b00010, let pcv be the permute control vector required to enable a right-indexed permute (*vpermr* or ***xxpermr***) to implement an expansion of the rightmost halfword elements of a source vector into the halfword elements of a result vector specified by the halfword-element mask in VSR[VRB+32].

If IMM=0b00011, let pcv be the permute control vector required to enable a right-indexed permute (*vpermr* or ***xxpermr***) to implement a compression of the sparse halfword elements in a source vector specified by the halfword-element mask in VSR[VRB+32] into the rightmost halfword elements of a result vector.

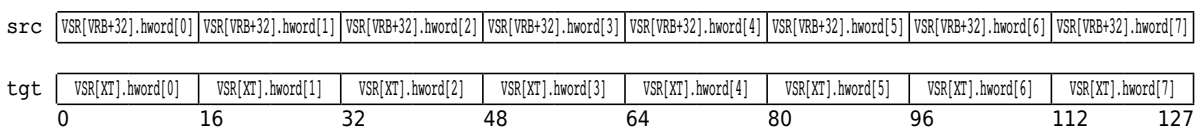
pcv is placed into VSR[XT].

Unused values of IMM are reserved.

Special Registers Altered:

None

VSR Data Layout for xxgenpcvhm



Programming Note

The following is an example of how a *Load VSX Vector Expand Halfword*, when using Big-Endian byte-ordering, can be emulated using *VSX Vector Generate PCV from Mask Halfword*.

```
xxgenpcvhm  vPCV, vMASK, 0b00000        // generates the required permute control vector
                                        // for Big-Endian expansion
vcntmbh     rN, vMASK, 0b1              // calculates N, number of true halfword-mask elements,
                                        // adjusted to # of bytes
lxvl        vLD, EA, rN                 // loads 2×N bytes

// Option 1: expand & merge
xxperm      vT, vLD, vT, vPCV           // perform the expansion,
                                        // specifying vT as 2nd source operand causes expanded
                                        // load data to be merged into VSR[vT]
// Option 2: expand & zero
xxperm      vT, vLD, vZERO, vPCV        // perform the expansion,
                                        // specifying vZERO (vector of 0s) as 2nd source operand
                                        // causes expanded load data to be placed into VSR[vT]
                                        // with other elements set to 0
```

The following is an example of how a *Load VSX Vector Expand Halfword*, when using Little-Endian byte-ordering, can be emulated using *VSX Vector Generate PCV from Mask Halfword*.

```
xxgenpcvhm  vPCV, vMASK, 0b00010        // generates the required permute control vector
                                        // for Little-Endian expansion
vcntmbh     rN, vMASK, 0b1              // calculates N, number of true halfword-mask elements,
                                        // adjusted to # of bytes
lxvl        vLD, EA, rN                 // loads 2×N bytes

// Option 1: expand & merge
xxpermr     vT, vLD, vT, vPCV           // perform the expansion,
                                        // specifying vT as 2nd source operand causes expanded
                                        // load data to be merged into VSR[vT]
// Option 2: expand & zero
xxpermr     vT, vLD, vZERO, vPCV        // perform the expansion,
                                        // specifying vZERO (vector of 0s) as 2nd source operand
                                        // causes expanded load data to be placed into VSR[vT]
                                        // with other elements set to 0
```

The following is an example of how a *Store VSX Vector Compress Halfword*, when using Big-Endian byte-ordering, can be emulated using *VSX Vector Generate PCV from Mask Halfword*.

```
xxgenpcvhm  vPCV, vMASK, 0b00001        // generates the required permute control vector
                                        // for Big-Endian compression
vcntmbh     rN, vMASK, 0b1              // calculates N, number of true halfword-mask elements,
                                        // adjusted to # of bytes
xxperm      vSD, vS, vS, vPCV           // perform the compression
stxvl       vSD, rEA, rN                // store 2×N bytes
```

The following is an example of how a *Store VSX Vector Compress Halfword*, when using Little-Endian byte-ordering, can be emulated using *VSX Vector Generate PCV from Mask Halfword*.

```
xxgenpcvhm  vPCV, vMASK, 0b00011        // generates the required permute control vector
                                        // for Little-Endian compression
vcntmbh     rN, vMASK, 0b1              // calculates N, number of true halfword-mask elements,
                                        // adjusted to # of bytes
xxpermr     vSD, vS, vS, vPCV           // perform the compression
stxvl       vSD, rEA, rN                // store 2×N bytes
```

VSX Vector Generate PCV from Word Mask X-form

xxgenpcvwm XT,VRB,IMM

60	T	IMM	VRB	948	TX
0	6	11	16	21	31

```
if MSR.VSX=0 then VSX_Unavailable()

XT ← 32×TX+T
if IMM=0b00000 then do  // Big-Endian expansion
   j ← 0
   do i = 0 to 3
      if VSR[VRB+32].word[i].bit[0]=1 then do
         VSR[XT].word[i].byte[0] ← 4×j + 0x00
         VSR[XT].word[i].byte[1] ← 4×j + 0x01
         VSR[XT].word[i].byte[2] ← 4×j + 0x02
         VSR[XT].word[i].byte[3] ← 4×j + 0x03
         j = j + 1
      end
      else do
         VSR[XT].word[i].byte[0] ← 4×i + 0x10
         VSR[XT].word[i].byte[1] ← 4×i + 0x11
         VSR[XT].word[i].byte[2] ← 4×i + 0x12
         VSR[XT].word[i].byte[3] ← 4×i + 0x13
      end
   end
end

else if IMM=0b00001 then do  // Big-Endian compression
   j ← 0
   do i = 0 to 3
      if VSR[VRB+32].word[i].bit[0]=1 then do
         VSR[XT].word[j].byte[0] ← 4×i + 0x00
         VSR[XT].word[j].byte[1] ← 4×i + 0x01
         VSR[XT].word[j].byte[2] ← 4×i + 0x02
         VSR[XT].word[j].byte[3] ← 4×i + 0x03
         j ← j + 1
      end
   end
   do i = j to 3
      VSR[XT].word[i] ← 0xUUUU_UUUU
   end
end

else if IMM=0b00010 then do  // Little-Endian expansion
   j ← 0
   do i = 0 to 3
      if VSR[VRB+32].word[3-i].bit[0]=1 then do
         VSR[XT].word[3-i].byte[3] ← 4×j + 0x00
         VSR[XT].word[3-i].byte[2] ← 4×j + 0x01
         VSR[XT].word[3-i].byte[1] ← 4×j + 0x02
         VSR[XT].word[3-i].byte[0] ← 4×j + 0x03
         j ← j + 1
      end
      else do
         VSR[XT].word[3-i].byte[3] ← 4×i + 0x10
         VSR[XT].word[3-i].byte[2] ← 4×i + 0x11
         VSR[XT].word[3-i].byte[1] ← 4×i + 0x12
         VSR[XT].word[3-i].byte[0] ← 4×i + 0x13
      end
   end
end
else if IMM=0b00011 then do  // Little-Endian compression
   j ← 0
   do i = 0 to 3
      if VSR[VRB+32].word[3-i].bit[0]=1 then do
         VSR[XT].word[3-j].byte[3] ← 4×i + 0x00
         VSR[XT].word[3-j].byte[2] ← 4×i + 0x01
         VSR[XT].word[3-j].byte[1] ← 4×i + 0x02
         VSR[XT].word[3-j].byte[0] ← 4×i + 0x03
         j ← j + 1
      end
   end
   do i = j to 3
         VSR[XT].word[3-i] ← 0xUUUU_UUUU
   end
end
```

Let XT be the value 32×TX + T.

If IMM=0b00000, let pcv be the permute control vector required to enable a left-indexed permute (*vperm* or ***xxperm***) to implement an expansion of the leftmost word elements of a source vector into the word elements of a result vector specified by the word-element mask in VSR[VRB+32].

If IMM=0b00001, let pcv be the permute control vector required to enable a left-indexed permute (*vperm* or ***xxperm***) to implement a compression of the sparse word elements in a source vector specified by the word-element mask in VSR[VRB+32] into the leftmost word elements of a result vector.

If IMM=0b00010, let pcv be the permute control vector required to enable a right-indexed permute (*vpermr* or ***xxpermr***) to implement an expansion of the rightmost word elements of a source vector into the word elements of a result vector specified by the word-element mask in VSR[VRB+32].

If IMM=0b00011, let pcv be the permute control vector required to enable a right-indexed permute (*vpermr* or ***xxpermr***) to implement a compression of the sparse word elements in a source vector specified by the word-element mask in VSR[VRB+32] into the rightmost word elements of a result vector.

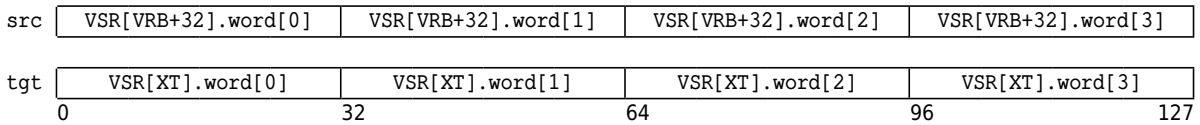
pcv is placed into VSR[XT].

Unused values of IMM are reserved.

Special Registers Altered:

None

VSR Data Layout for xxgenpcvwm



Programming Note

The following is an example of how a *Load VSX Vector Expand Word*, when using Big-Endian byte-ordering, can be emulated using *VSX Vector Generate PCV from Mask Word*.

```
xxgenpcvwm  vPCV, vMASK, 0b00000        // generates the required permute control vector
                                        // for Big-Endian expansion
vcntmbw     rN, vMASK, 0b1              // calculates N, number of true word-mask elements,
                                        // adjusted to # of bytes
lxvl        vLD, EA, rN                 // loads 4×N bytes

// Option 1: expand & merge
xxperm      vT, vLD, vT, vPCV           // perform the expansion,
                                        // specifying vT as 2nd source operand causes expanded
                                        // load data to be merged into VSR[vT]
// Option 2: expand & zero
xxperm      vT, vLD, vZERO, vPCV        // perform the expansion,
                                        // specifying vZERO (vector of 0s) as 2nd source operand
                                        // causes expanded load data to be placed into VSR[vT]
                                        // with other elements set to 0
```

The following is an example of how a *Load VSX Vector Expand Word*, when using Little-Endian byte-ordering, can be emulated using *VSX Vector Generate PCV from Mask Word*.

```
xxgenpcvwm  vPCV, vMASK, 0b00010        // generates the required permute control vector
                                        // for Little-Endian expansion
vcntmbw     rN, vMASK, 0b1              // calculates N, number of true word-mask elements,
                                        // adjusted to # of bytes
lxvl        vLD, EA, rN                 // loads 4×N bytes

// Option 1: expand & merge
xxpermr     vT, vLD, vT, vPCV           // perform the expansion,
                                        // specifying vT as 2nd source operand causes expanded
                                        // load data to be merged into VSR[vT]
// Option 2: expand & zero
xxpermr     vT, vLD, vZERO, vPCV        // perform the expansion,
                                        // specifying vZERO (vector of 0s) as 2nd source operand
                                        // causes expanded load data to be placed into VSR[vT]
                                        // with other elements set to 0
```

The following is an example of how a *Store VSX Vector Compress Word*, when using Big-Endian byte-ordering, can be emulated using *VSX Vector Generate PCV from Mask Word*.

```
xxgenpcvwm  vPCV, vMASK, 0b00001        // generates the required permute control vector
                                        // for Big-Endian compression
vcntmbw     rN, vMASK, 0b1              // calculates N, number of true word-mask elements,
                                        // adjusted to # of bytes
xxperm      vSD, vS, vS, vPCV           // perform the compression
stxvl       vSD, rEA, rN                // store 4×N bytes
```

The following is an example of how a *Store VSX Vector Compress Word*, when using Little-Endian byte-ordering, can be emulated using *VSX Vector Generate PCV from Mask Word*.

```
xxgenpcvwm  vPCV, vMASK, 0b00011        // generates the required permute control vector
                                        // for Little-Endian compression
vcntmbw     rN, vMASK, 0b1              // calculates N, number of true word-mask elements,
                                        // adjusted to # of bytes
xxpermr     vSD, vS, vS, vPCV           // perform the compression
stxvl       vSD, rEA, rN                // store 4×N bytes
```

7.6.2.15 VSX Vector Load Special Value Instruction

Load VSX Vector Special Value Quadword X-form

lxvkq XT,UIM

60	T	31	UIM	360	TX
0	6	11	16	21	31

```
if MSR.VSX=0 then VSX_Unavailable()

if UIM=0b00001 then VSR[32×TX+T] ← 0x3FFF_0000_0000_0000_0000_0000_0000_0000  /* QP +1.0  */
if UIM=0b00010 then VSR[32×TX+T] ← 0x4000_0000_0000_0000_0000_0000_0000_0000  /* QP +2.0  */
if UIM=0b00011 then VSR[32×TX+T] ← 0x4000_8000_0000_0000_0000_0000_0000_0000  /* QP +3.0  */
if UIM=0b00100 then VSR[32×TX+T] ← 0x4001_0000_0000_0000_0000_0000_0000_0000  /* QP +4.0  */
if UIM=0b00101 then VSR[32×TX+T] ← 0x4001_4000_0000_0000_0000_0000_0000_0000  /* QP +5.0  */
if UIM=0b00110 then VSR[32×TX+T] ← 0x4001_8000_0000_0000_0000_0000_0000_0000  /* QP +6.0  */
if UIM=0b00111 then VSR[32×TX+T] ← 0x4001_C000_0000_0000_0000_0000_0000_0000  /* QP +7.0  */
if UIM=0b01000 then VSR[32×TX+T] ← 0x7FFF_0000_0000_0000_0000_0000_0000_0000  /* QP +Inf  */
if UIM=0b01001 then VSR[32×TX+T] ← 0x7FFF_8000_0000_0000_0000_0000_0000_0000  /* QP dQNaN */
if UIM=0b10000 then VSR[32×TX+T] ← 0x8000_0000_0000_0000_0000_0000_0000_0000  /* QP -0.0  */
if UIM=0b10001 then VSR[32×TX+T] ← 0xBFFF_0000_0000_0000_0000_0000_0000_0000  /* QP -1.0  */
if UIM=0b10010 then VSR[32×TX+T] ← 0xC000_0000_0000_0000_0000_0000_0000_0000  /* QP -2.0  */
if UIM=0b10011 then VSR[32×TX+T] ← 0xC000_8000_0000_0000_0000_0000_0000_0000  /* QP -3.0  */
if UIM=0b10100 then VSR[32×TX+T] ← 0xC001_0000_0000_0000_0000_0000_0000_0000  /* QP -4.0  */
if UIM=0b10101 then VSR[32×TX+T] ← 0xC001_4000_0000_0000_0000_0000_0000_0000  /* QP -5.0  */
if UIM=0b10110 then VSR[32×TX+T] ← 0xC001_8000_0000_0000_0000_0000_0000_0000  /* QP -6.0  */
if UIM=0b10111 then VSR[32×TX+T] ← 0xC001_C000_0000_0000_0000_0000_0000_0000  /* QP -7.0  */
if UIM=0b11000 then VSR[32×TX+T] ← 0xFFFF_0000_0000_0000_0000_0000_0000_0000  /* QP -Inf  */
```

Let XT be the value 32×TX + T.

UIM specifies one of a set of common values that is placed into VSR[XT]. Unspecified values of UIM are reserved.

Special Registers Altered:

None

VSR Data Layout for lxvkq

7.6.2.16 VSX Vector Test Least-Significant Bit by Byte Instruction

VSX Vector Test Least-Significant Bit by Byte XX2-form

xvtlsbb BF,XB

60	BF	//	2	B	475	BX	/
0	6	9	11	16	21	30	31

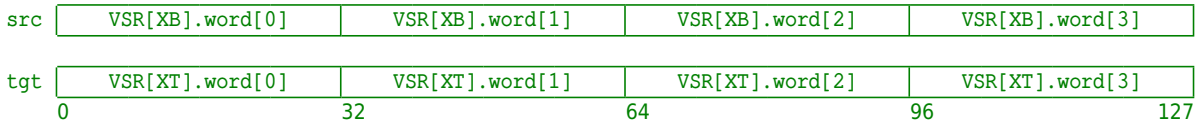
```
if MSR.VSX=0 then VSX_Unavailable()

ALL_TRUE  ← 1
ALL_FALSE ← 1

do i = 0 to 15
   ALL_TRUE  ← ALL_TRUE  & (VSR[XB].byte[i].bit[7]=1)
   ALL_FALSE ← ALL_FALSE & (VSR[XB].byte[i].bit[7]=0)
end

CR.field[BF] ← ALL_TRUE || 0 || ALL_FALSE || 0;
```

Set CR field BF to indicate if bit 7 of every byte element in VSR[XB] is equal to 1 (ALL_TRUE) or equal to 0 (ALL_FALSE).

Special Registers Altered:

Register	*Field(s)*
CR	field BF

VSR Data Layout for xvtlsbb



Programming Note

This instruction following any *Vector Compare* provides the ability to direct the summary status of the *Vector Compare* to any CR field, not just CR field 6 when Rc=1.

Appendix A. Suggested Floating-Point Models

A.1 Floating-Point Round to Single-Precision Model

The following describes algorithmically the operation of the *Floating Round to Single-Precision* instruction.

```
If (FRB)1:11 < 897 and (FRB)1:63 > 0 then
   Do
      If FPSCRUE = 0 then goto Disabled Exponent Underflow
      If FPSCRUE = 1 then goto Enabled Exponent Underflow
   End

If (FRB)1:11 > 1150 and (FRB)1:11 < 2047 then
   Do
      If FPSCROE = 0 then goto Disabled Exponent Overflow
      If FPSCROE = 1 then goto Enabled Exponent Overflow
   End

If (FRB)1:11 > 896 and (FRB)1:11 < 1151 then goto Normal Operand

If (FRB)1:63 = 0 then goto Zero Operand

If (FRB)1:11 = 2047 then
   Do
      If (FRB)12:63 = 0 then goto Infinity Operand
      If (FRB)12 = 1 then goto QNaN Operand
      If (FRB)12 = 0 and (FRB)13:63 > 0 then goto SNaN Operand
   End
```

Disabled Exponent Underflow:

```
sign ← (FRB)0
If (FRB)1:11 = 0 then
   Do
      exp ← −1022
      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
Denormalize operand:
   G || R || X ← 0b000
   Do while exp < −126
      exp ← exp + 1
      frac0:52 || G || R || X ← 0b0 || frac0:52 || G || (R | X)
   End
FPSCRUX ← (frac24:52 || G || R || X) > 0
Round Single(sign,exp,frac0:52,G,R,X)
FPSCRXX ← FPSCRXX | FPSCRFI
```

```
If frac0:52 = 0 then
   Do
      FRT0 ← sign
      FRT1:63 ← 0
      If sign = 0 then FPSCRFPRF ← ''+ zero''
      If sign = 1 then FPSCRFPRF ← ''− zero''
   End
If frac0:52 > 0 then
   Do
      If frac0 = 1 then
         Do
            If sign = 0 then FPSCRFPRF ← ''+ normal number''
            If sign = 1 then FPSCRFPRF ← ''− normal number''
         End
      If frac0 = 0 then
         Do
            If sign = 0 then FPSCRFPRF ← ''+ denormalized number''
            If sign = 1 then FPSCRFPRF ← ''− 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 | FPSCRFI
exp ← exp + 192
FRT0 ← sign
FRT1:11 ← exp + 1023
FRT12:63 ← frac1:52
If sign = 0 then FPSCRFPRF ← ''+ normal number''
If sign = 1 then FPSCRFPRF ← ''− normal number''
Done
```

Disabled Exponent Overflow:

```
FPSCROX ← 1
If FPSCRRN = 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 FPSCRFPRF ← ''+ infinity''
         If (FRB)0 = 1 then FPSCRFPRF ← ''− infinity''
      End
   If FPSCRRN = 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 FPSCRFPRF ← ''+ normal number''
         If (FRB)0 = 1 then FPSCRFPRF ← ''− normal number''
      End
   If FPSCRRN = 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 FPSCRFPRF ← ''+ infinity''
         If (FRB)0 = 1 then FPSCRFPRF ← ''− normal number''
      End
   If FPSCRRN = 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 FPSCRFPRF ← ''+ normal number''
         If (FRB)0 = 1 then FPSCRFPRF ← ''− infinity''
      End
   FPSCRFR ← undefined
   FPSCRFI ← 1
   FPSCRXX ← 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)
   FPSCRXX ← FPSCRXX | FPSCRFI
Enabled Overflow:
   FPSCROX ← 1
   exp ← exp − 192
   FRT0 ← sign
   FRT1:11 ← exp + 1023
   FRT12:63 ← frac1:52
   If sign = 0 then FPSCRFPRF ← ''+ normal number''
   If sign = 1 then FPSCRFPRF ← ''− normal number''
   Done
```

Zero Operand:

```
   FRT ← (FRB)
   If (FRB)0 = 0 then FPSCRFPRF ← ''+ zero''
   If (FRB)0 = 1 then FPSCRFPRF ← ''− zero''
   FPSCRFR FI ← 0b00
   Done
```

Infinity Operand:

```
   FRT ← (FRB)
   If (FRB)0 = 0 then FPSCRFPRF ← ''+ infinity''
   If (FRB)0 = 1 then FPSCRFPRF ← ''− infinity''
   FPSCRFR FI ← 0b00
   Done
```

QNaN Operand:

```
FRT ← (FRB)0:34 || 29 0
FPSCRFPRF ← ''QNaN''
FPSCRFR FI ← 0b00
Done
```

SNaN Operand:

```
FPSCRVXSNAN ← 1
If FPSCRVE = 0 then
   Do
      FRT0:11 ← (FRB)0:11
      FRT12 ← 1
      FRT13:63 ← (FRB)13:34 || 29 0
      FPSCRFPRF ← ''QNaN''
   End
FPSCRFR FI ← 0b00
Done
```

Normal Operand:

```
sign ← (FRB)0
exp ← (FRB)1:11 − 1023
frac0:52 ← 0b1 || (FRB)12:63
Round Single(sign,exp,frac0:52,0,0,0)
FPSCRXX ← FPSCRXX | FPSCRFI
If exp > 127 and FPSCROE = 0 then go to Disabled Exponent Overflow
If exp > 127 and FPSCROE = 1 then go to Enabled Overflow
FRT0 ← sign
FRT1:11 ← exp + 1023
FRT12:63 ← frac1:52
If sign = 0 then FPSCRFPRF ← ''+ normal number''
If sign = 1 then FPSCRFPRF ← ''− normal number''
Done
```

Round Single(sign,exp,frac0:52,G,R,X):

```
inc ← 0
lsb ← frac23
gbit ← frac24
rbit ← frac25
xbit ← (frac26:52||G||R||X)≠0
If FPSCRRN = 0b00 then        /* Round to Nearest */
   Do         /* comparisons ignore u bits */
   If sign || lsb || gbit || rbit || xbit = 0bu11uu then inc ← 1
   If sign || lsb || gbit || rbit || xbit = 0bu011u then inc ← 1
   If sign || lsb || gbit || rbit || xbit = 0bu01u1 then inc ← 1
   End
If FPSCRRN = 0b10 then        /* Round toward + Infinity */
   Do         /* comparisons ignore u bits */
   If sign || lsb || gbit || rbit || xbit = 0b0u1uu then inc ← 1
   If sign || lsb || gbit || rbit || xbit = 0b0uu1u then inc ← 1
   If sign || lsb || gbit || rbit || xbit = 0b0uuu1 then inc ← 1
   End
If FPSCRRN = 0b11 then        /* Round toward − Infinity */
   Do         /* comparisons ignore u bits */
   If sign || lsb || gbit || rbit || xbit = 0b1u1uu then inc ← 1
   If sign || lsb || gbit || rbit || xbit = 0b1uu1u then inc ← 1
   If sign || lsb || gbit || rbit || xbit = 0b1uuu1 then inc ← 1
   End
frac0:23 ← frac0:23 + inc
If carry_out = 1 then
   Do
```

```
        frac0:23 ← 0b1 || frac0:22
        exp ← exp + 1
    End
frac24:52 ← 29 0
FPSCRFR ← inc
FPSCRFI ← gbit | rbit | xbit
Return
```

A.2 Floating-Point Convert to Integer Model

The following describes algorithmically the operation of the *Floating Convert To Integer* instructions.

```
if Floating Convert To Integer Word then do
   round_mode     ← FPSCR_RN
   tgt_precision ← ''32-bit signed integer''
end

if Floating Convert To Integer Word Unsigned then do
   round_mode     ← FPSCR_RN
   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_RN
   tgt_precision ← ''64-bit signed integer''
end

if Floating Convert To Integer Doubleword Unsigned then do
   round_mode     ← FPSCR_RN
   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 || 11^0   /* normal */
if (FRB)_1:11 = 0 then frac_0:64 ← 0b00 || (FRB)_12:63 || 11^0   /* 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 frac0:64 ← ¬frac0:64 + 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 | FPSCRFI

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
FPSCRFPRF ← 0bUUUUU
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 = 0b0U1UU then inc ← 1
   if sign || frac64 || gbit || rbit || xbit = 0b0UU1U then inc ← 1
   if sign || frac64 || gbit || rbit || xbit = 0b0UUU1 then inc ← 1
end
if round_mode = 0b11 then do   /* Round toward -Infinity */
   if sign || frac64 || gbit || rbit || xbit = 0b1U1UU then inc ← 1
   if sign || frac64 || gbit || rbit || xbit = 0b1UU1U then inc ← 1
   if sign || frac64 || gbit || rbit || xbit = 0b1UUU1 then inc ← 1
end
frac0:64 ← frac0:64 + inc
FPSCRFR ← inc
FPSCRFI ← gbit | rbit | xbit
return
```

Infinity Operand:

```
FPSCRFR ← 0b0
FPSCRFI ← 0b0
FPSCRVXCVI ← 0b1
if FPSCRVE = 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 = ''32-bit unsigned integer'' then do
         if sign=0 then FRT ← 0xUUUU_UUUU_FFFF_FFFF
         if sign=1 then FRT ← 0xUUUU_UUUU_0000_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 = ''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 ← 0bUUUUU
   end
   done
```

SNaN Operand:

```
   FPSCR_FR ← 0b0
   FPSCR_FI ← 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
      FPSCR_FPRF ← 0bUUUUU
   end
   done
```

QNaN Operand:

```
   FPSCR_FR ← 0b0
   FPSCR_FI ← 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
      FPSCR_FPRF ← 0bUUUUU
   end
   done
```

Large Operand:

```
   FPSCR_FR ← 0b0
   FPSCR_FI ← 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 ← 0xUUUU_UUUU_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 ← 0bUUUUU
end
done
```

A.3 Floating-Point Convert from Integer Model

The following describes algorithmically the operation of the *Floating Convert From Integer* instructions.

```
if  Floating Convert From Integer Doubleword  then do
   tgt_precision ← ''double-precision''
   sign     ← (FRB)0
   exp      ← 63
   frac0:63  ← (FRB)
end
if  Floating Convert From Integer Doubleword Single  then do
   tgt_precision ← ''single-precision''
   sign     ← (FRB)0
   exp      ← 63
   frac0:63  ← (FRB)
end
if  Floating Convert From Integer Doubleword Unsigned  then do
   tgt_precision ← ''double-precision''
   sign     ← 0
   exp      ← 63
   frac0:63  ← (FRB)
end
if  Floating Convert From Integer Doubleword Unsigned Single  then do
   tgt_precision ← ''single-precision''
   sign     ← 0
   exp      ← 63
   frac0:63  ← (FRB)
end

if frac0:63 = 0 then go to Zero Operand
if sign = 1 then frac0:63 ← ¬frac0:63 + 1

/* do the loop 0 times if (FRB) = max negative 64-bit integer or */
/*                     if (FRB) = max unsigned 64-bit integer    */
do while frac0 = 0
   frac0:63 ← frac1:63 || 0b0
   exp ← exp - 1
end

Round Float( sign, exp, frac0:63, RN )
if sign = 0 then FPSCRFPRF ← ''+normal number''
if sign = 1 then FPSCRFPRF ← ''-normal number''
FRT0 ← sign
FRT1:11 ← exp + 1023   /* exp + bias */
FRT12:63 ← frac1:52
done
```

Zero Operand:

```
FPSCRFR ← 0b00
FPSCRFI ← 0b00
FPSCRFPRF ← ''+ zero''
FRT ← 0x0000_0000_0000_0000
done
```

Round Float(sign, exp, frac$_{0:63}$, 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 = 0bU11UU then inc ← 1
   if sign || lsb || gbit || rbit || xbit = 0bU011U then inc ← 1
   if sign || lsb || gbit || rbit || xbit = 0bU01U1 then inc ← 1
end
if round_mode = 0b10 then do               /* Round toward + Infinity */
   if sign || lsb || gbit || rbit || xbit = 0b0U1UU then inc ← 1
   if sign || lsb || gbit || rbit || xbit = 0b0UU1U then inc ← 1
   if sign || lsb || gbit || rbit || xbit = 0b0UUU1 then inc ← 1
end
if round_mode = 0b11 then do               /* Round toward − Infinity */
   if sign || lsb || gbit || rbit || xbit = 0b1U1UU then inc ← 1
   if sign || lsb || gbit || rbit || xbit = 0b1UU1U then inc ← 1
   if sign || lsb || gbit || rbit || xbit = 0b1UUU1 then inc ← 1
end

if tgt_precision = ``single-precision'' then
   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
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 */

sign ← (FRB)0
exp ← (FRB)1:11 - 1023 /* exp - bias */
frac0:52 ← 0b1 || (FRB)12:63
gbit || rbit || xbit ← 0b000

Do i = 1, 52 - exp
   frac0:52 || gbit || rbit || xbit ← 0b0 || frac0:52 || gbit || (rbit | xbit)
End

Round Integer (sign, frac0:52, gbit, rbit, xbit)

Do i = 2, 52 - exp
   frac0:52 ← frac1:52 || 0b0
End

If frac0 = 1, then exp ← exp + 1
Else frac0:52 ← frac1:52 || 0b0

FRT0 ← sign
FRT1:11 ← exp + 1023
FRT12:63 ← frac1:52

If (FRT)0 = 0 then FPSCRFPRF ← ''+ normal number''
Else FPSCRFPRF ← ''- normal number''
FPSCRFR FI  ← 0b00
Done
```

Round Integer(sign, frac0:52, gbit, rbit, xbit):

```
inc ← 0
If inst = Floating Round to Integer Nearest then        /* ties away from zero */
   Do   /* comparisons ignore u bits */
      If sign || frac52 || gbit || rbit || xbit = 0buu1uu then inc ← 1
   End
If inst = Floating Round to Integer Plus then
   Do   /* comparisons ignore u bits */
      If sign || frac52 || gbit || rbit || xbit = 0b0u1uu then inc ← 1
      If sign || frac52 || gbit || rbit || xbit = 0b0uu1u then inc ← 1
      If sign || frac52 || gbit || rbit || xbit = 0b0uuu1 then inc ← 1
  End
If inst = Floating Round to Integer Minus then
   Do   /* comparisons ignore u bits */
      If sign || frac52 || gbit || rbit || xbit = 0b1u1uu then inc ← 1
      If sign || frac52 || gbit || rbit || xbit = 0b1uu1u then inc ← 1
      If sign || frac52 || gbit || rbit || xbit = 0b1uuu1 then inc ← 1
   End
frac0:52  ← frac0:52 + inc
Return
```

Infinity Operand:

```
FRT  ← (FRB)
```

```
If (FRB)0 = 0 then FPSCRFPRF ← ''+ infinity''
If (FRB)0 = 1 then FPSCRFPRF ← ''- infinity''
FPSCRFR FI  ← 0b00
Done
```

SNaN Operand:

```
FPSCRVXSNAN  ← 1
If FPSCRVE = 0 then
   Do
      FRT  ← (FRB)
      FRT12  ← 1
      FPSCRFPRF  ← ''QNaN''
   End
FPSCRFR FI  ← 0b00
Done
```

QNaN Operand:

```
FRT  ← (FRB)
FPSCRFPRF  ← ''QNaN''
FPSCRFR FI  ← 0b00
Done
```

Zero Operand:

```
If (FRB)0 = 0 then
   Do
      FRT ← 0x0000_0000_0000_0000
      FPSCRFPRF  ← ''+ zero''
   End
Else
   Do
      FRT ← 0x8000_0000_0000_0000
      FPSCRFPRF  ← ''- zero''
   End
FPSCRFR FI  ← 0b00
Done
```

Small Operand:

```
If inst = Floating Round to Integer Nearest and (FRB)1:11 < 1022 then goto Zero Operand
If inst = Floating Round to Integer Toward Zero then goto Zero Operand
If inst = Floating Round to Integer Plus and (FRB)0 = 1 then goto Zero Operand
If inst = Floating Round to Integer Minus and (FRB)0 = 0 then goto Zero Operand

If (FRB)0 = 0 then
   Do
      FRT  ← 0x3FF0_0000_0000_0000 /* value = 1.0 */
      FPSCRFPRF  ← ''+ normal number''
   End
Else
   Do
      FRT  ← 0xBFF0_0000_0000_0000 /* value = -1.0 */
      FPSCRFPRF  ← ''- normal number''
   End
FPSCRFR FI  ← 0b00
Done
```

Large Operand:

```
FRT ← (FRB)
If FRT0 = 0 then FPSCRFPRF ← ''+ normal number''
Else FPSCRFPRF  ← ''- normal number''
FPSCRFR FI  ← 0b00
Done
```

Appendix B. Densely Packed Decimal

The trailing significand field of the decimal floating-point data format is encoded using Densely Packed Decimal (DPD). DPD encoding is a compression technique which supports the representation of decimal integers of arbitrary length. Translation operates on three Binary Coded Decimal (BCD) digits at a time compressing the 12 bits into 10 bits with an algorithm that can be applied or reversed using simple Boolean operations. In the following examples, a 3-digit BCD number is represented as (abcd)(efgh)(ijkm), a 10-bit DPD number is represented as (pqr)(stu)(v)(wxy), and the Boolean operations, & (AND), | (OR), and ¬ (NOT) are used.

B.1 BCD-to-DPD Translation

The translation from a 3-digit BCD number to a 10-bit DPD can be performed through the following Boolean operations.

```
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 = (¬e & j & ¬i) | (e & i) | a
x = (¬a & k & ¬i) | (a & i) | e
y = m
```

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.

aei	pqr stu v wxy
000	bcd fgh 0 jkm
001	bcd fgh 1 00m
010	bcd jkh 1 01m
011	bcd 10h 1 11m
100	jkd fgh 1 10m
101	fgd 01h 1 11m
110	jkd 00h 1 11m
111	00d 11h 1 11m

The full translation of a 3-digit BCD number (000 - 999) to a 10-bit DPD is shown in Table B.1 on page 952, 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.

```
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 = r
e = (v & ¬w & x) | (s & v & w & x) |
    (¬t & v & x & w)
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
```

Alternatively, the following table can be used to perform

the translation. A combination of five bits in the DPD encoding (leftmost column) are used to specify a translation to the 3-digit BCD encoding. Dashes (-) in the table are don't cares, and can be either one or zero.

vwxst	abcd	efgh	ijkm
0----	0pqr	0stu	0wxy
100--	0pqr	0stu	100y
101--	0pqr	100u	0sty
110--	100r	0stu	0pqy
11100	100r	100u	0pqy
11101	100r	0pqu	100y
11110	0pqr	100u	100y
11111	100r	100u	100y

The full translation of the 10-bit DPD to a 3-digit BCD number is shown in Table B.2 on page 953. 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.

DPD Code	BCD Value	DPD Code	BCD Value
0x06E	888	0x0EE	988
(0x16E)		(0x1EE)	
(0x26E)		(0x2EE)	
(0x36E)		(0x3EE)	
0x06F	889	0x0EF	989
(0x16F)		(0x1EF)	
(0x26F)		(0x2EF)	
(0x36F)		(0x3EF)	
0x07E	898	0x0FE	998
(0x17E)		(0x1FE)	
(0x27E)		(0x2FE)	
(0x37E)		(0x3FE)	
0x07F	899	0x0FF	999
(0x17F)		(0x1FF)	
(0x27F)		(0x2FF)	
(0x37F)		(0x3FF)	

	0	1	2	3	4	5	6	7	8	9
00_	000	001	002	003	004	005	006	007	008	009
01_	010	011	012	013	014	015	016	017	018	019
02_	020	021	022	023	024	025	026	027	028	029
03_	030	031	032	033	034	035	036	037	038	039
04_	040	041	042	043	044	045	046	047	048	049
05_	050	051	052	053	054	055	056	057	058	059
06_	060	061	062	063	064	065	066	067	068	069
07_	070	071	072	073	074	075	076	077	078	079
08_	00A	00B	02A	02B	04A	04B	06A	06B	04E	04F
09_	01A	01B	03A	03B	05A	05B	07A	07B	05E	05F
10_	080	081	082	083	084	085	086	087	088	089
11_	090	091	092	093	094	095	096	097	098	099
12_	0A0	0A1	0A2	0A3	0A4	0A5	0A6	0A7	0A8	0A9
13_	0B0	0B1	0B2	0B3	0B4	0B5	0B6	0B7	0B8	0B9
14_	0C0	0C1	0C2	0C3	0C4	0C5	0C6	0C7	0C8	0C9
15_	0D0	0D1	0D2	0D3	0D4	0D5	0D6	0D7	0D8	0D9
16_	0E0	0E1	0E2	0E3	0E4	0E5	0E6	0E7	0E8	0E9
17_	0F0	0F1	0F2	0F3	0F4	0F5	0F6	0F7	0F8	0F9
18_	08A	08B	0AA	0AB	0CA	0CB	0EA	0EB	0CE	0CF
19_	09A	09B	0BA	0BB	0DA	0DB	0FA	0FB	0DE	0DF
20_	100	101	102	103	104	105	106	107	108	109
21_	110	111	112	113	114	115	116	117	118	119
22_	120	121	122	123	124	125	126	127	128	129
23_	130	131	132	133	134	135	136	137	138	139
24_	140	141	142	143	144	145	146	147	148	149
25_	150	151	152	153	154	155	156	157	158	159
26_	160	161	162	163	164	165	166	167	168	169
27_	170	171	172	173	174	175	176	177	178	179
28_	10A	10B	12A	12B	14A	14B	16A	16B	14E	14F
29_	11A	11B	13A	13B	15A	15B	17A	17B	15E	15F
30_	180	181	182	183	184	185	186	187	188	189
31_	190	191	192	193	194	195	196	197	198	199
32_	1A0	1A1	1A2	1A3	1A4	1A5	1A6	1A7	1A8	1A9
33_	1B0	1B1	1B2	1B3	1B4	1B5	1B6	1B7	1B8	1B9
34_	1C0	1C1	1C2	1C3	1C4	1C5	1C6	1C7	1C8	1C9
35_	1D0	1D1	1D2	1D3	1D4	1D5	1D6	1D7	1D8	1D9
36_	1E0	1E1	1E2	1E3	1E4	1E5	1E6	1E7	1E8	1E9
37_	1F0	1F1	1F2	1F3	1F4	1F5	1F6	1F7	1F8	1F9
38_	18A	18B	1AA	1AB	1CA	1CB	1EA	1EB	1CE	1CF
39_	19A	19B	1BA	1BB	1DA	1DB	1FA	1FB	1DE	1DF
40_	200	201	202	203	204	205	206	207	208	209
41_	210	211	212	213	214	215	216	217	218	219
42_	220	221	222	223	224	225	226	227	228	229
43_	230	231	232	233	234	235	236	237	238	239
44_	240	241	242	243	244	245	246	247	248	249
45_	250	251	252	253	254	255	256	257	258	259
46_	260	261	262	263	264	265	266	267	268	269
47_	270	271	272	273	274	275	276	277	278	279
48_	20A	20B	22A	22B	24A	24B	26A	26B	24E	24F
49_	21A	21B	23A	23B	25A	25B	27A	27B	25E	25F

	0	1	2	3	4	5	6	7	8	9
50_	280	281	282	283	284	285	286	287	288	289
51_	290	291	292	293	294	295	296	297	298	299
52_	2A0	2A1	2A2	2A3	2A4	2A5	2A6	2A7	2A8	2A9
53_	2B0	2B1	2B2	2B3	2B4	2B5	2B6	2B7	2B8	2B9
54_	2C0	2C1	2C2	2C3	2C4	2C5	2C6	2C7	2C8	2C9
55_	2D0	2D1	2D2	2D3	2D4	2D5	2D6	2D7	2D8	2D9
56_	2E0	2E1	2E2	2E3	2E4	2E5	2E6	2E7	2E8	2E9
57_	2F0	2F1	2F2	2F3	2F4	2F5	2F6	2F7	2F8	2F9
58_	28A	28B	2AA	2AB	2CA	2CB	2EA	2EB	2CE	2CF
59_	29A	29B	2BA	2BB	2DA	2DB	2FA	2FB	2DE	2DF
60_	300	301	302	303	304	305	306	307	308	309
61_	310	311	312	313	314	315	316	317	318	319
62_	320	321	322	323	324	325	326	327	328	329
63_	330	331	332	333	334	335	336	337	338	339
64_	340	341	342	343	344	345	346	347	348	349
65_	350	351	352	353	354	355	356	357	358	359
66_	360	361	362	363	364	365	366	367	368	369
67_	370	371	372	373	374	375	376	377	378	379
68_	30A	30B	32A	32B	34A	34B	36A	36B	34E	34F
69_	31A	31B	33A	33B	35A	35B	37A	37B	35E	35F
70_	380	381	382	383	384	385	386	387	388	389
71_	390	391	392	393	394	395	396	397	398	399
72_	3A0	3A1	3A2	3A3	3A4	3A5	3A6	3A7	3A8	3A9
73_	3B0	3B1	3B2	3B3	3B4	3B5	3B6	3B7	3B8	3B9
74_	3C0	3C1	3C2	3C3	3C4	3C5	3C6	3C7	3C8	3C9
75_	3D0	3D1	3D2	3D3	3D4	3D5	3D6	3D7	3D8	3D9
76_	3E0	3E1	3E2	3E3	3E4	3E5	3E6	3E7	3E8	3E9
77_	3F0	3F1	3F2	3F3	3F4	3F5	3F6	3F7	3F8	3F9
78_	38A	38B	3AA	3AB	3CA	3CB	3EA	3EB	3CE	3CF
79_	39A	39B	3BA	3BB	3DA	3DB	3FA	3FB	3DE	3DF
80_	00C	00D	10C	10D	20C	20D	30C	30D	02E	02F
81_	01C	01D	11C	11D	21C	21D	31C	31D	03E	03F
82_	02C	02D	12C	12D	22C	22D	32C	32D	12E	12F
83_	03C	03D	13C	13D	23C	23D	33C	33D	13E	13F
84_	04C	04D	14C	14D	24C	24D	34C	34D	22E	22F
85_	05C	05D	15C	15D	25C	25D	35C	35D	23E	23F
86_	06C	06D	16C	16D	26C	26D	36C	36D	32E	32F
87_	07C	07D	17C	17D	27C	27D	37C	37D	33E	33F
88_	00E	00F	10E	10F	20E	20F	30E	30F	06E	06F
89_	01E	01F	11E	11F	21E	21F	31E	31F	07E	07F
90_	08C	08D	18C	18D	28C	28D	38C	38D	0AE	0AF
91_	09C	09D	19C	19D	29C	29D	39C	39D	0BE	0BF
92_	0AC	0AD	1AC	1AD	2AC	2AD	3AC	3AD	1AE	1AF
93_	0BC	0BD	1BC	1BD	2BC	2BD	3BC	3BD	1BE	1BF
94_	0CC	0CD	1CC	1CD	2CC	2CD	3CC	3CD	2AE	2AF
95_	0DC	0DD	1DC	1DD	2DC	2DD	3DC	3DD	2BE	2BF
96_	0EC	0ED	1EC	1ED	2EC	2ED	3EC	3ED	3AE	3AF
97_	0FC	0FD	1FC	1FD	2FC	2FD	3FC	3FD	3BE	3BF
98_	08E	08F	18E	18F	28E	28F	38E	38F	0EE	0EF
99_	09E	09F	19E	19F	29E	29F	39E	39F	0FE	0FF

Table B.1: BCD-to-DPD translation

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00_	000	001	002	003	004	005	006	007	008	009	080	081	800	801	880	881
01_	010	011	012	013	014	015	016	017	018	019	090	091	810	811	890	891
02_	020	021	022	023	024	025	026	027	028	029	082	083	820	821	808	809
03_	030	031	032	033	034	035	036	037	038	039	092	093	830	831	818	819
04_	040	041	042	043	044	045	046	047	048	049	084	085	840	841	088	089
05_	050	051	052	053	054	055	056	057	058	059	094	095	850	851	098	099
06_	060	061	062	063	064	065	066	067	068	069	086	087	860	861	888	889
07_	070	071	072	073	074	075	076	077	078	079	096	097	870	871	898	899
08_	100	101	102	103	104	105	106	107	108	109	180	181	900	901	980	981
09_	110	111	112	113	114	115	116	117	118	119	190	191	910	911	990	991
0A_	120	121	122	123	124	125	126	127	128	129	182	183	920	921	908	909
0B_	130	131	132	133	134	135	136	137	138	139	192	193	930	931	918	919
0C_	140	141	142	143	144	145	146	147	148	149	184	185	940	941	188	189
0D_	150	151	152	153	154	155	156	157	158	159	194	195	950	951	198	199
0E_	160	161	162	163	164	165	166	167	168	169	186	187	960	961	988	989
0F_	170	171	172	173	174	175	176	177	178	179	196	197	970	971	998	999
10_	200	201	202	203	204	205	206	207	208	209	280	281	802	803	882	883
11_	210	211	212	213	214	215	216	217	218	219	290	291	812	813	892	893
12_	220	221	222	223	224	225	226	227	228	229	282	283	822	823	828	829
13_	230	231	232	233	234	235	236	237	238	239	292	293	832	833	838	839
14_	240	241	242	243	244	245	246	247	248	249	284	285	842	843	288	289
15_	250	251	252	253	254	255	256	257	258	259	294	295	852	853	298	299
16_	260	261	262	263	264	265	266	267	268	269	286	287	862	863	(888)	(889)
17_	270	271	272	273	274	275	276	277	278	279	296	297	872	873	(898)	(899)
18_	300	301	302	303	304	305	306	307	308	309	380	381	902	903	982	983
19_	310	311	312	313	314	315	316	317	318	319	390	391	912	913	992	993
1A_	320	321	322	323	324	325	326	327	328	329	382	383	922	923	928	929
1B_	330	331	332	333	334	335	336	337	338	339	392	393	932	933	938	939
1C_	340	341	342	343	344	345	346	347	348	349	384	385	942	943	388	389
1D_	350	351	352	353	354	355	356	357	358	359	394	395	952	953	398	399
1E_	360	361	362	363	364	365	366	367	368	369	386	387	962	963	(988)	(989)
1F_	370	371	372	373	374	375	376	377	378	379	396	397	972	973	(998)	(999)
20_	400	401	402	403	404	405	406	407	408	409	480	481	804	805	884	885
21_	410	411	412	413	414	415	416	417	418	419	490	491	814	815	894	895
22_	420	421	422	423	424	425	426	427	428	429	482	483	824	825	848	849
23_	430	431	432	433	434	435	436	437	438	439	492	493	834	835	858	859
24_	440	441	442	443	444	445	446	447	448	449	484	485	844	845	488	489
25_	450	451	452	453	454	455	456	457	458	459	494	495	854	855	498	499
26_	460	461	462	463	464	465	466	467	468	469	486	487	864	865	(888)	(889)
27_	470	471	472	473	474	475	476	477	478	479	496	497	874	875	(898)	(899)
28_	500	501	502	503	504	505	506	507	508	509	580	581	904	905	984	985
29_	510	511	512	513	514	515	516	517	518	519	590	591	914	915	994	995
2A_	520	521	522	523	524	525	526	527	528	529	582	583	924	925	948	949
2B_	530	531	532	533	534	535	536	537	538	539	592	593	934	935	958	959
2C_	540	541	542	543	544	545	546	547	548	549	584	585	944	945	588	589
2D_	550	551	552	553	554	555	556	557	558	559	594	595	954	955	598	599
2E_	560	561	562	563	564	565	566	567	568	569	586	587	964	965	(988)	(989)
2F_	570	571	572	573	574	575	576	577	578	579	596	597	974	975	(998)	(999)
30_	600	601	602	603	604	605	606	607	608	609	680	681	806	807	886	887
31_	610	611	612	613	614	615	616	617	618	619	690	691	816	817	896	897
32_	620	621	622	623	624	625	626	627	628	629	682	683	826	827	868	869
33_	630	631	632	633	634	635	636	637	638	639	692	693	836	837	878	879
34_	640	641	642	643	644	645	646	647	648	649	684	685	846	847	688	689
35_	650	651	652	653	654	655	656	657	658	659	694	695	856	857	698	699
36_	660	661	662	663	664	665	666	667	668	669	686	687	866	867	(888)	(889)
37_	670	671	672	673	674	675	676	677	678	679	696	697	876	877	(898)	(899)
38_	700	701	702	703	704	705	706	707	708	709	780	781	906	907	986	987
39_	710	711	712	713	714	715	716	717	718	719	790	791	916	917	996	997
3A_	720	721	722	723	724	725	726	727	728	729	782	783	926	927	968	969
3B_	730	731	732	733	734	735	736	737	738	739	792	793	936	937	978	979
3C_	740	741	742	743	744	745	746	747	748	749	784	785	946	947	788	789
3D_	750	751	752	753	754	755	756	757	758	759	794	795	956	957	798	799
3E_	760	761	762	763	764	765	766	767	768	769	786	787	966	967	(988)	(989)
3F_	770	771	772	773	774	775	776	777	778	779	796	797	976	977	(998)	(999)

Table B.2: DPD-to-BCD translation

Appendix C. Assembler Extended Mnemonics

In order to make assembler language programs simpler to write and easier to understand, a set of extended mnemonics and symbols is provided that defines simple shorthand for the most frequently used forms of *Branch Conditional*, *Compare*, *Trap*, *Rotate and Shift*, and certain other instructions.

Assemblers should provide the extended mnemonics and symbols listed here, and may provide others.

C.1 Symbols

The following symbols are defined for use in instructions (basic or extended mnemonics) that specify a Condition Register field or a Condition Register bit. The first five (lt, ..., un) identify a bit number within a CR field. The remainder (cr0, ..., cr7) identify a CR field. An expression in which a CR field symbol is multiplied by 4 and then added to a bit-number-within-CR-field symbol and 32 can be used to identify a CR bit.

Symbol	Value	Meaning
lt	0	Less than
gt	1	Greater than
eq	2	Equal
so	3	Summary overflow
un	3	Unordered (after floating-point comparison)
cr0	0	CR Field 0
cr1	1	CR Field 1
cr2	2	CR Field 2
cr3	3	CR Field 3
cr4	4	CR Field 4
cr5	5	CR Field 5
cr6	6	CR Field 6
cr7	7	CR Field 7

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***], ***bclr***[***l***], ***bcctr***[***l***]) with the appropriate operands.

C.2.2 Simple Branch Mnemonics

Instructions using one of the mnemonics in Table C.1 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.

Branch Semantics	**LR not Set**				**LR Set**			
	bc **Relative**	***bca*** **Absolute**	***bclr*** **To LR**	***bcctr*** **To CTR**	***bcl*** **Relative**	***bcla*** **Absolute**	***bclrl*** **To LR**	***bcctrl*** **To CTR**
Branch unconditionally	–	–	blr	bctr	–	–	blrl	bctrl
Branch if CR_{BI}=1	bt	bta	btlr	btctr	btl	btla	btlrl	btctrl
Branch if CR_{BI}=0	bf	bfa	bflr	bfctr	bfl	bfla	bflrl	bfctrl
Decrement CTR, branch if CTR nonzero	bdnz	bdnza	bdnzlr	–	bdnzl	bdnzla	bdnzlrl	–
Decrement CTR, branch if CTR nonzero and CR_{BI}=1	bdnzt	bdnzta	bdnztlr	–	bdnztl	bdnztla	bdnztlrl	–
Decrement CTR, branch if CTR nonzero and CR_{BI}=0	bdnzf	bdnzfa	bdnzflr	–	bdnzfl	bdnzfla	bdnzflrl	–
Decrement CTR, branch if CTR zero	bdz	bdza	bdzlr	–	bdzl	bdzla	bdzlrl	–
Decrement CTR, branch if CTR zero and CR_{BI}=1	bdzt	bdzta	bdztlr	–	bdztl	bdztla	bdztlrl	–
Decrement CTR, branch if CTR zero and CR_{BI}=0	bdzf	bdzfa	bdzflr	–	bdzfl	bdzfla	bdzflrl	–

Table C.1: Simple branch mnemonics

Examples

1. Decrement CTR and branch if it is still nonzero (closure of a loop controlled by a count loaded into CTR).
   bdnz target (equivalent to: bc 16,0,target)
2. Same as (1) but branch only if CTR is nonzero and condition in CR0 is “equal”.
   bdnzt target (equivalent to: bc 8,2,target)
3. Same as (2), but “equal” condition is in CR5.
   bdnzt 4×cr5+eq,target (equivalent to: bc 8,22,target)
4. Branch if bit 59 of CR is 0.
   bf 27,target (equivalent to: bc 4,27,target)
5. Same as (4), but set the Link Register. This is a form of conditional “call”.
   bfl 27,target (equivalent to: bcl 4,27,target)

C.2.3 Branch Mnemonics Incorporating Conditions

In the mnemonics defined in Table C.2, the test of a bit in a Condition Register field is encoded in the mnemonic.

Instructions using the mnemonics in Table C.2 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 ***bclr***[***l***] or ***bcctr***[***l***] and the BH operand is specified.

A standard set of codes has been adopted for the most common combinations of branch conditions.

Code	Meaning
lt	Less than
le	Less than or equal
eq	Equal
ge	Greater than or equal
gt	Greater than
nl	Not less than
ne	Not equal
ng	Not greater than
so	Summary overflow
ns	Not summary overflow
un	Unordered (after floating-point comparison)
nu	Not unordered (after floating-point comparison)

These codes are reflected in the mnemonics shown in Table C.2.

Branch Semantics	LR not Set				LR Set			
	bc **Relative**	***bca*** **Absolute**	***bclr*** **To LR**	***bcctr*** **To CTR**	***bcl*** **Relative**	***bcla*** **Absolute**	***bclrl*** **To LR**	***bcctrl*** **To CTR**
Branch if less than	blt	blta	bltlr	bltctr	bltl	bltla	bltlrl	bltctrl
Branch if less than or equal	ble	blea	blelr	blectr	blel	blela	blelrl	blectrl
Branch if equal	beq	beqa	beqlr	beqctr	beql	beqla	beqlrl	beqctrl
Branch if greater than or equal	bge	bgea	bgelr	bgectr	bgel	bgela	bgelrl	bgectrl
Branch if greater than	bgt	bgta	bgtlr	bgtctr	bgtl	bgtla	bgtlrl	bgtctrl
Branch if not less than	bnl	bnla	bnllr	bnlctr	bnll	bnlla	bnllrl	bnlctrl
Branch if not equal	bne	bnea	bnelr	bnectr	bnel	bnela	bnelrl	bnectrl
Branch if not greater than	bng	bnga	bnglr	bngctr	bngl	bngla	bnglrl	bngctrl
Branch if summary overflow	bso	bsoa	bsolr	bsoctr	bsol	bsola	bsolrl	bsoctrl
Branch if not summary overflow	bns	bnsa	bnslr	bnsctr	bnsl	bnsla	bnslrl	bnsctrl
Branch if unordered	bun	buna	bunlr	bunctr	bunl	bunla	bunlrl	bunctrl
Branch if not unordered	bnu	bnua	bnulr	bnuctr	bnul	bnula	bnulrl	bnuctrl

Table C.2: Branch mnemonics incorporating conditions

Examples

1. Branch if CR0 reflects condition “not equal”.

   bne target (equivalent to: bc 4,2,target)

2. Same as (1), but condition is in CR3.

   bne cr3,target (equivalent to: bc 4,14,target)

3. Branch to an absolute target if CR4 specifies “greater than”, setting the Link Register. This is a form of conditional “call”.

   bgtla cr4,target (equivalent to: bcla 12,17,target)

4. Same as (3), but target address is in the Count Register.

   bgtctrl cr4 (equivalent to: bcctrl 12,17,0)

C.2.4 Branch Prediction

Software can use the “at” bits of *Branch Conditional* instructions to provide a hint to the processor about the behavior of the branch. If, for a given such instruction, the branch is almost always taken or almost always not taken, a suffix can be added to the mnemonic indicating the value to be used for the “at” bits.

+ Predict branch to be taken (at=0b11)
– Predict branch not to be taken (at=0b10)

Such a suffix can be added to any *Branch Conditional* mnemonic, either basic or extended, that tests either the Count Register or a CR bit (but not both). Assemblers should use 0b00 as the default value for the “at” bits, indicating that software has offered no prediction.

Examples

1. Branch if CR0 reflects condition “less than”, specifying that the branch should be predicted to be taken.

   ```
   blt+    target
   ```

2. Same as (1), but target address is in the Link Register and the branch should be predicted not to be taken.

   ```
   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.

Operation	Extended Mnemonic	Equivalent to
Condition Register set	crset bx	creqv bx,bx,bx
Condition Register clear	crclr bx	crxor bx,bx,bx
Condition Register move	crmove bx,by	cror bx,by,by
Condition Register not	crnot bx,by	crnor bx,by,by

Table C.3: Condition Register logical mnemonics

The symbols defined in Section C.1 can be used to identify the Condition Register bits.

Examples

1. Set CR bit 57.

   crset 25 (equivalent to: creqv 25,25,25)

2. Clear the SO bit of CR0.

   crclr so (equivalent to: crxor 3,3,3)

3. Same as (2), but SO bit to be cleared is in CR3.

   crclr 4×cr3+so (equivalent to: crxor 15,15,15)

4. Invert the EQ bit.

   crnot 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.

   crnot 4×cr5+eq,4×cr4+eq (equivalent to: crnor 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:  addic.  Rx,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

Operation	Extended Mnemonic	Equivalent to
Compare doubleword immediate	cmpdi bf,ra,si	cmpi bf,1,ra,si
Compare doubleword	cmpd bf,ra,rb	cmp bf,1,ra,rb
Compare logical doubleword immediate	cmpldi bf,ra,ui	cmpli bf,1,ra,ui
Compare logical doubleword	cmpld bf,ra,rb	cmpl bf,1,ra,rb

Table C.4: Doubleword compare mnemonics

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

Operation	Extended Mnemonic	Equivalent to
Compare word immediate	cmpwi bf,ra,si	cmpi bf,0,ra,si
Compare word	cmpw bf,ra,rb	cmp bf,0,ra,rb
Compare logical word immediate	cmplwi bf,ra,ui	cmpli bf,0,ra,ui
Compare logical word	cmplw bf,ra,rb	cmpl bf,0,ra,rb

Table C.5: Word compare mnemonics

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: cmpl 0,0,Rx,Ry)

C.6 Trap Mnemonics

The mnemonics defined in Table C.6 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.

Code	Meaning	TO encoding	$<$	$>$	$=$	$<^u$	$>^u$
lt	Less than	16	1	0	0	0	0
le	Less than or equal	20	1	0	1	0	0
eq	Equal	4	0	0	1	0	0
ge	Greater than or equal	12	0	1	1	0	0
gt	Greater than	8	0	1	0	0	0
nl	Not less than	12	0	1	1	0	0
ne	Not equal	24	1	1	0	0	0
ng	Not greater than	20	1	0	1	0	0
llt	Logically less than	2	0	0	0	1	0
lle	Logically less than or equal	6	0	0	1	1	0
lge	Logically greater than or equal	5	0	0	1	0	1
lgt	Logically greater than	1	0	0	0	0	1
lnl	Logically not less than	5	0	0	1	0	1
lng	Logically not greater than	6	0	0	1	1	0
u	Unconditionally with parameters	31	1	1	1	1	1
(none)	Unconditional	31	1	1	1	1	1

These codes are reflected in the mnemonics shown in Table C.6.

Trap Semantics	64-bit Comparison		32-bit Comparison	
	tdi **Immediate**	***td*** **Register**	***twi*** **Immediate**	***tw*** **Register**
Trap unconditionally	-	-	-	trap
Trap unconditionally with parameters	tdui	tdu	twui	twu
Trap if less than	tdlti	tdlt	twlti	twlt
Trap if less than or equal	tdlei	tdle	twlei	twle
Trap if equal	tdeqi	tdeq	tweqi	tweq
Trap if greater than or equal	tdgei	tdge	twgei	twge
Trap if greater than	tdgti	tdgt	twgti	twgt
Trap if not less than	tdnli	tdnl	twnli	twnl
Trap if not equal	tdnei	tdne	twnei	twne
Trap if not greater than	tdngi	tdng	twngi	twng
Trap if logically less than	tdllti	tdllt	twllti	twllt
Trap if logically less than or equal	tdllei	tdlle	twllei	twlle
Trap if logically greater than or equal	tdlgei	tdlge	twlgei	twlge
Trap if logically greater than	tdlgti	tdlgt	twlgti	twlgt
Trap if logically not less than	tdlnli	tdlnl	twlnli	twlnl
Trap if logically not greater than	tdlngi	tdlng	twlngi	twlng

Table C.6: Trap mnemonics

Examples

1. Trap if register Rx is not 0.

```
tdnei   Rx,0        (equivalent to:   tdi   24,Rx,0)
```

2. Same as (1), but comparison is to register Ry.

```
tdne    Rx,Ry       (equivalent to:   td    24,Rx,Ry)
```

3. Trap if bits 32:63 of register Rx, considered as a 32-bit quantity, are logically greater than 0x7FF.

   twlgti Rx,0x7FF (equivalent to: twi 1,Rx,0x7FF)

4. Trap unconditionally.

   trap (equivalent to: tw 31,0,0)

5. Trap unconditionally with register parameters Rx and Ry

   tdu Rx,Ry (equivalent to: td 31,Rx,Ry)

C.7 Integer Select Mnemonics

The mnemonics defined in Table C.7, “Integer Select mnemonics,” on page 961 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.

Code	Meaning
lt	Less than
eq	Equal
gt	Greater than

These codes are reflected in the mnemonics shown in Table C.7.

Select semantics	***isel* extended mnemonic**
Integer Select if less than	isellt
Integer Select if equal	iseleq
Integer Select if greater than	iselgt

Table C.7: Integer Select mnemonics

Examples

1. Set register Rx to Ry if the LT bit is set in CR0, and to Rz otherwise.

   isellt Rx,Ry,Rz (equivalent to: isel Rx,Ry,Rz,0)

2. Set register Rx to Ry if the GT bit is set in CR0, and to Rz otherwise.

   iselgt Rx,Ry,Rz (equivalent to: isel Rx,Ry,Rz,1)

3. Set register Rx to Ry if the EQ bit is set in CR0, and to Rz otherwise.

   iseleq Rx,Ry,Rz (equivalent to: isel Rx,Ry,Rz,2)

C.8 Rotate and Shift Mnemonics

The *Rotate and Shift* instructions provide powerful and general ways to manipulate register contents, but can be difficult to understand. Extended mnemonics are provided that allow some of the simpler operations to be coded easily.

Mnemonics are provided for the following types of operation.

Extract Select a field of n bits starting at bit position b in the source register; left or right justify this field in the target register; clear all other bits of the target register to 0.

Insert Select a left-justified or right-justified field of n bits in the source register; insert this field starting at bit position b of the target register; leave other bits of the target register unchanged. (No extended mnemonic is provided for insertion of a left-justified field when operating on doublewords, because such an insertion requires more than one instruction.)

Rotate Rotate the contents of a register right or left n bits without masking.

Shift Shift the contents of a register right or left n bits, clearing vacated bits to 0 (logical shift).

Clear Clear the leftmost or rightmost n bits of a register to 0.

Clear left and shift left
Clear the leftmost b bits of a register, then shift the register left by n bits. This operation can be used to scale a (known nonnegative) array index by the width of an element.

C.8.1 Operations on Doublewords

All these mnemonics can be coded with a final "." to cause the Rc bit to be set in the underlying instruction.

Operation	Extended Mnemonic	Equivalent to
Extract and left justify immediate	extldi ra,rs,n,b (n $>$ 0)	rldicr ra,rs,b,n−1
Extract and right justify immediate	extrdi ra,rs,n,b (n $>$ 0)	rldicl ra,rs,b+n,64−n
Insert from right immediate	insrdi ra,rs,n,b (n $>$ 0)	rldimi ra,rs,64−(b+n),b
Rotate left immediate	rotldi ra,rs,n	rldicl ra,rs,n,0
Rotate right immediate	rotrdi ra,rs,n	rldicl ra,rs,64−n,0
Rotate left	rotld ra,rs,rb	rldcl ra,rs,rb,0
Shift left immediate	sldi ra,rs,n (n $<$ 64)	rldicr ra,rs,n,63−n
Shift right immediate	srdi ra,rs,n (n $<$ 64)	rldicl ra,rs,64−n,n
Clear left immediate	clrldi ra,rs,n (n $<$ 64)	rldicl ra,rs,0,n
Clear right immediate	clrrdi ra,rs,n (n $<$ 64)	rldicr ra,rs,0,63−n
Clear left and shift left immediate	clrlsldi ra,rs,b,n (n $\leq$ b $<$ 64)	rldic ra,rs,n,b−n

Table C.8: Doubleword rotate and shift mnemonics

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)

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.

Operation	Extended Mnemonic	Equivalent to
Extract and left justify immediate	extlwi ra,rs,n,b (n $>$ 0)	rlwinm ra,rs,b,0,n$-$1
Extract and right justify immediate	extrwi ra,rs,n,b (n $>$ 0)	rlwinm ra,rs,b+n,32$-$n,31
Insert from left immediate	inslwi ra,rs,n,b (n $>$ 0)	rlwimi ra,rs,32$-$b,b,(b+n)$-$1
Insert from right immediate	insrwi ra,rs,n,b (n $>$ 0)	rlwimi ra,rs,32$-$(b+n),b,(b+n)$-$1
Rotate left immediate	rotlwi ra,rs,n	rlwinm ra,rs,n,0,31
Rotate right immediate	rotrwi ra,rs,n	rlwinm ra,rs,32$-$n,0,31
Rotate left	rotlw ra,rs,rb	rlwnm ra,rs,rb,0,31
Shift left immediate	slwi ra,rs,n (n $<$ 32)	rlwinm ra,rs,n,0,31$-$n
Shift right immediate	srwi ra,rs,n (n $<$ 32)	rlwinm ra,rs,32$-$n,n,31
Clear left immediate	clrlwi ra,rs,n (n $<$ 32)	rlwinm ra,rs,0,n,31
Clear right immediate	clrrwi ra,rs,n (n $<$ 32)	rlwinm ra,rs,0,0,31$-$n
Clear left and shift left immediate	clrlslwi ra,rs,b,n (n $\leq$ b $<$ 32)	rlwinm ra,rs,n,b$-$n,31$-$n

Table C.9: Word rotate and shift mnemonics

Examples

1. Extract the sign bit (bit 32) of register Ry and place the result right-justified into register Rx.

   extrwi Rx,Ry,1,0 (equivalent to: rlwinm 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.

Special Purpose Register	Move To SPR		Move From SPR	
	Extended	Equivalent to	Extended	Equivalent to
XER	mtxer Rx	mtspr 1,Rx	mfxer Rx	mfspr Rx,1
DSCR	mtudscr Rx	mtspr 3,Rx	mfudscr Rx	mfspr Rx,3
LR	mtlr Rx	mtspr 8,Rx	mflr Rx	mfspr Rx,8
CTR	mtctr Rx	mtspr 9,Rx	mfctr Rx	mfspr Rx,9
AMR	mtuamr Rx	mtspr 13,Rx	mfuamr Rx	mfspr Rx,13
CTRL	–	–	mfctrl Rx	mfspr Rx,136
VRSAVE	mtvrsave Rx	mtspr 256,Rx	mfvrsave Rx	mfspr Rx,256
SPRG3	–	–	mfusprg3 Rx	mfspr Rx,259
TB	–	–	mftb Rx	mftb Rx,268 mfspr Rx,268
TBU	–	–	mftbu Rx	mftb Rx,269 mfspr Rx,269
SIER	–	–	mfusier Rx	mfspr Rx,768
MMCR2	mtummcr2 Rx	mtspr 769,Rx	mfummcr2 Rx	mfspr Rx,769
MMCRA	mtummcra Rx	mtspr 770,Rx	mfummcra Rx	mfspr Rx,770
PMC1	mtupmc1 Rx	mtspr 771,Rx	mfupmc1 Rx	mfspr Rx,771
PMC2	mtupmc2 Rx	mtspr 772,Rx	mfupmc2 Rx	mfspr Rx,772
PMC3	mtupmc3 Rx	mtspr 773,Rx	mfupmc3 Rx	mfspr Rx,773
PMC4	mtupmc4 Rx	mtspr 774,Rx	mfupmc4 Rx	mfspr Rx,774
PMC5	mtupmc5 Rx	mtspr 775,Rx	mfupmc5 Rx	mfspr Rx,775
PMC6	mtupmc6 Rx	mtspr 776,Rx	mfupmc6 Rx	mfspr Rx,776
MMCR0	mtummcr0 Rx	mtspr 779,Rx	mfummcr0 Rx	mfspr Rx,779
SIAR	–	–	mfusiar Rx	mfspr Rx,780
SDAR	–	–	mfusdar Rx	mfspr Rx,781
MMCR1	–	–	mfummcr1 Rx	mfspr Rx,782
BESCRS	mtbescrs Rx	mtspr 800,Rx	mfbescrs Rx	mfspr Rx,800
BESCRSU	mtbescrsu Rx	mtspr 801,Rx	mfbescrsu Rx	mfspr Rx,801
BESCRR	mtbescrr Rx	mtspr 802,Rx	mfbescrr Rx	mfspr Rx,802
BESCRRU	mtbescrru Rx	mtspr 803,Rx	mfbescrru Rx	mfspr Rx,803
EBBHR	mtebbhr Rx	mtspr 804,Rx	mfebbhr Rx	mfspr Rx,804
EBBRR	mtebbrr Rx	mtspr 805,Rx	mfebbrr Rx	mfspr Rx,805
BESCR	mtbescr Rx	mtspr 806,Rx	mfbescr Rx	mfspr Rx,806
TAR	mttar Rx	mtspr 815,Rx	mftar Rx	mfspr Rx,815
PPR	mtppr Rx	mtspr 896,Rx	mfppr Rx	mfspr Rx,896
PPR32	mtppr32 Rx	mtspr 898,Rx	mfppr32 Rx	mfspr Rx,898

Table C.10: Extended mnemonics for moving to/from an SPR

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 desireable. 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)

To avoid certain security vulnerabilities, it is sometimes desirable to constrain the order in which instructions are executed at certain points in a program. An extended mnemonic is provided for a form of the *Or Immediate* instruction that serves this purpose. See Section 9.2.1 of Book III.

exser		(equivalent to:	ori	31,31,0)

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 ***addpcis*** instruction can be used to load the next instruction address into a register. An extended mnemonics is provided to perform this operation.

lnia	Rx	(equivalent to:	addpcis	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.

la	Rx,D(Ry)	(equivalent to:	addi	Rx,Ry,D)

The ***la*** mnemonic is useful for obtaining the address of a variable specified by name, allowing the Assembler to supply the base register number and compute the displacement. If the variable v is located at offset Dv bytes from the address in register Rv, and the Assembler has been told to use register Rv as a base for references to the data structure containing v, then the following line causes the address of v to be loaded into register Rx.

la	Rx,v	(equivalent to:	addi	Rx,Rv,Dv)

Move Register

Several Power ISA instructions can be coded in a way such that they simply copy the contents of one register to another. An extended mnemonic is provided to convey the idea that no computation is being performed but merely data movement (from one register to another).

The following instruction copies the contents of register Ry to register Rx. This mnemonic can be coded with a final "." to cause the Rc bit to be set in the underlying instruction.

mr	Rx,Ry	(equivalent to:	or	Rx,Ry,Ry)

Complement Register

Several Power ISA instructions can be coded in a way such that they complement the contents of one register and place the result into another register. An extended mnemonic is provided that allows this operation to be coded easily.

The following instruction complements the contents of register Ry and places the result into register Rx. This mnemonic can be coded with a final "." to cause the Rc bit to be set in the underlying instruction.

```
not     Rx,Ry                   (equivalent to:   nor     Rx,Ry,Ry)
```

Move To/From Condition Register

This mnemonic permits copying the contents of the low-order 32 bits of a GPR to the Condition Register, using the same style as the ***mfcr*** instruction.

```
mtcr    Rx                      (equivalent to:   mtcrf   0xFF,Rx)
```

The following instructions may generate either the (old) ***mtcrf*** or ***mfcr*** instructions or the (new) ***mtocrf*** or ***mfocrf*** instruction, respectively, depending on the target machine type assembler parameter.

```
mtcrf   FXM,Rx
mfcr    Rx
```

All three extended mnemonics in this subsection are being phased out. In future assemblers the form "mtcr Rx" may not exist, and the ***mtcrf*** and ***mfcr*** mnemonics may generate the old form instructions (with bit 11 = 0) regardless of the target machine type assembler parameter, or may cease to exist.

Book II

Power ISA Virtual Environment Architecture

Chapter 1. Storage Model

1.1 Definitions

The following definitions, in addition to those specified in Book I, are used in this Book. In these definitions, "*Load* instruction" includes the *Cache Management* and other instructions that are stated in the instruction descriptions to be "treated as a *Load*", and similarly for "*Store* instruction".

- **system**
  A combination of processors, storage, and associated mechanisms that is capable of executing programs. Sometimes the reference to system includes services provided by the privileged software.
- **main storage**
  The level of storage hierarchy in which all storage state is visible to all processors and mechanisms in the system.
- **normal memory**
  Coherently-accessed, well-behaved regions of main storage that hold supervisor software and data and general purpose applications and data. This is in contrast with regions of main storage associated with I/O interfaces or devices such as accelerators or attached to other systems. Normal memory is assumed to have the following storage control attributes (see Section 1.6): not Write Through Required, not Caching Inhibited, Memory Coherence Required, and not Guarded.
- **control memory**
  Regions of main storage that are associated with individual devices or groups of similar devices and used to deliver control information, such as commands, to the respective devices. Control memory can be accessed only as the target of a data transfer (see Section 1.7.1.1).
- **persistent storage**
  Storage that retains its contents when power is removed, and is "behind" certain regions of main storage, that is not Caching Inhibited, in the sense that a store (to a location in such a region of main storage) will get to main storage before it gets to persistent storage.
- **primary cache**
  The level of cache closest to the processor.
- **secondary cache**
  After the primary cache, the next closest level of cache to the processor.
- **instruction storage**
  The view of storage as seen by the mechanism that fetches instructions.
- **data storage**
  The view of storage as seen by a *Load* or *Store* instruction.
- **program order**
  The execution of instructions in the order required by the sequential execution model. (See Section 2.2 of Book I.) A ***dcbz*** instruction that modifies storage which contains instructions has the same effect with respect to the sequential execution model as a *Store* instruction as described there.)
  For the instructions and facilities defined in this Book, there are two additional exceptions to the sequential execution model that the processor obeys beyond those described in Section 2.2 of Book I.
  - an event-based branch occurs (see Chapter 6)
  - the BHRB is read (see Section 7.2)
- **event-based exception**
  An unusual condition, or external signal, that sets a status bit in the BESCR and may or may not cause an event-based branch, depending upon whether event-based branches are enabled.
- **storage location**
  A contiguous sequence of one or more bytes in storage. When used in association with a specific instruction or the instruction fetching mechanism, the length of the sequence of one or more bytes is typically implied by the operation. In other uses, it may refer more abstractly to a group of bytes which share common storage attributes.
- **storage access**
  An access to a storage location. There are three (mutually exclusive) kinds of storage access.

  - **data access**

    An access to the storage location specified by a *Load* or *Store* instruction, or, if the access is performed "out-of-order" (see Section 6.5 of Book III), an access to a storage location as if it were the storage location specified by a *Load* or *Store* instruction.
  - **instruction fetch**

    An access for the purpose of fetching an instruction.
  - **implicit access**

    An access by the processor for the purpose of finding the address translation tables, translating an address, or recording reference and change information (see Book III).

- **caused by, associated with**
  - **caused by**

    A storage access is said to be caused by an instruction if the instruction is a *Load* or *Store* and the access (data access) is to the storage location specified by the instruction.
  - **associated with**

    A storage access is said to be associated with an instruction if the access is for the purpose of fetching the instruction (instruction fetch), or is a data access caused by the instruction, or is an implicit access that occurs as a side effect of fetching or executing the instruction.

- **metadata**

  Companion data associated with a storage location. In addition to the data that is loaded into a target register or stored from a source register, a storage location may be associated with additional control information. The granularity, meaning, and treatment of the control information may vary based on the type of storage access involved and on the state of the process making the access. Unless otherwise stated or obvious from context, loads ignore the metadata and stores zero the metadata.

  Note that not all storage locations have associated metadata. The absence of associated metadata does not necessarily prevent successful completion of an instruction that specifies the treatment of metadata. Unless otherwise stated or obvious from context, metadata associated with a store is lost, and a load will have its metadata set to zero if the storage location has no associated metadata.

- **prefetched instructions**

  Instructions for which a copy of the instruction has been fetched from instruction storage, but the instruction has not yet been executed.

- **uniprocessor**

  A system that contains one processor.

- **multiprocessor**

  A system that contains two or more processors.

- **shared storage multiprocessor**

  A multiprocessor that contains some common storage, which all the processors in the system can access.

- **performed**

  A load or instruction fetch by a processor or mechanism (P1) is performed with respect to any processor or mechanism (P2) when the value to be returned by the load or instruction fetch can no longer be changed by a store by P2. A store by P1 is performed with respect to P2 when a load by P2 from the location accessed by the store will return the value stored (or a value stored subsequently). An instruction cache block invalidation by P1 is performed with respect to P2 when the instruction that requested the invalidation has caused the specified block, if present, to be made invalid in P2's instruction cache, and similarly for a data cache block invalidation.

  The preceding definitions apply regardless of whether P1 and P2 are the same entity.

- **page (virtual page)**

  2^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.

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, ag-

gressive implementations of the architecture can permit intervals of time during which different copies of a storage location have different values. This chapter describes features of the Power ISA that enable programmers to write correct programs for this storage model.

1.3 Virtual Storage

The Power ISA system implements a virtual storage model for applications. This means that a combination of hardware and software can present a storage model that allows applications to exist within a "virtual" address space larger than either the effective address space or the real address space.

Each program can access 2^{64} bytes of "effective address" (EA) space, subject to limitations imposed by the operating system. In a typical Power ISA system, each program's EA space is a subset of a larger "virtual address" (VA) space managed by the operating system.

Each effective address is translated to a real address (i.e., to an address of a byte in real storage or on an I/O device) before being used to access storage. The hardware accomplishes this, using the address translation mechanism described in Book III. The operating system manages the real (physical) storage resources of the system, by setting up the tables and other information used by the hardware address translation mechanism.

In general, real storage may not be large enough to map all the virtual pages used by the currently active applications. With support provided by hardware, the operating system can attempt to use the available real pages to map a sufficient set of virtual pages of the applications. If a sufficient set is maintained, "paging" activity is minimized. If not, performance degradation is likely.

The operating system can support restricted access to virtual pages (including read/write, read only, and no access; see Book III), based on system standards (e.g., program code might be read only) and application requests.

1.4 Single-Copy Atomicity

An access is *single-copy atomic*, or simply *atomic*, if it is always performed in its entirety with no visible fragmentation. Atomic accesses are thus serialized: each happens in its entirety in some order, even when that order is not specified in the program or enforced between processors.

The access caused by an instruction other than a *Load/Store Multiple* or *Move Assist* instruction is guaranteed to be atomic if the storage operand is not larger than a doubleword and is aligned (see Section 1.10.1 of Book I). For *Load* (*Store*) *Multiple Word* instructions, this section applies as if the instruction were the corresponding sequence of *Load Word and Zero* (*Store Word*) instructions. Thus, for example, the access to each word specified by a *Load/Store Multiple Word* instruction is guaranteed to be atomic if the storage operand is aligned.

Quadword accesses with aligned storage operands are guaranteed to be atomic when caused by the following instructions.

- ***lq***
- ***stq***
- ***lqarx***
- ***stqcx.***

Quadword atomicity applies only to storage that is neither Write Through Required nor Caching Inhibited. The cases described above are the only cases in which the access to the storage operand is guaranteed to be atomic. For example, the access caused by the following instructions is not guaranteed to be atomic.

- any *Load* or *Store* instruction for which the storage operand is unaligned
- ***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 *Move Assist* instruction and one of the following conditions is satisfied, the non-atomic access is performed as described in the corresponding list item. The first list item matching a given situation applies.

- The storage operand is at least 16 bytes long, has length that is an integral multiple of eight, and is doubleword-aligned:
  the access is performed as a set of disjoint atomic accesses each of which consists of one or more aligned doublewords.
- The storage operand is at least eight bytes long, has length that is an integral multiple of four, 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, has length that is an integral multiple of two, and is halfword-aligned:
  the access is performed as a set of disjoint atomic accesses each of which consists of one or more aligned halfwords.

In all other cases the number, length, and alignment of the component disjoint atomic accesses are implementation-dependent. In all cases the relative order in which the component disjoint atomic accesses are performed is implementation-dependent.

The results for several combinations of loads and stores to the same or overlapping locations are described below.

1. When two processors perform atomic stores to locations that do not overlap, and no other stores are performed to those locations, the contents of those locations are the same as if the two stores were performed by a single processor.

2. When two processors perform atomic stores to the same storage location, and no other store is performed to that location, the contents of that location are the result stored by one of the processors.
3. When two processors perform stores that have the same target location and are not guaranteed to be atomic, and no other store is performed to that location, the result is some combination of the bytes stored by both processors.
4. When two processors perform stores to overlapping locations, and no other store is performed to those locations, the result is some combination of the bytes stored by the processors to the overlapping bytes. The portions of the locations that do not overlap contain the bytes stored by the processor storing to the location.
5. When a processor performs an atomic store to a location, a second processor performs an atomic load from that location, and no other store is performed to that location, the value returned by the load is the contents of the location before the store or the contents of the location after the store.
6. When a load and a store with the same target location can be performed simultaneously, and the accesses are not guaranteed to be atomic, and no other store is performed to that location, the value returned by the load is some combination of the contents of the location before the store and the contents of the location after the store.

Engineering Note

Atomicity of storage accesses is provided by the processor in conjunction with the storage subsystem. The processor must provide a storage subsystem interface that is sufficient to allow a storage subsystem to meet the atomicity requirements specified here.

For initial hardware debug it is often useful to run with cache disabled. In some ways cache disabled mode is similar to Caching Inhibited storage. Although quadword atomicity is not required for storage that is Caching Inhibited, support for quadword atomicity with cache disabled should be considered.

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 (***dcbt***, ***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 ***dcbfl***)

A write to main storage caused by a ***dcbst*** or ***dcbf*** instruction has updated main storage when a load by any given processor, that is satisfied from main storage from the location accessed by the write, will return the value written (or a value written or stored subsequently), and similarly for the store caused by a Store instruction.

1.6 Storage Control Attributes

Some operating systems may provide a means to allow programs to specify the storage control attributes described in this section. Because the support provided for these attributes by the operating system may vary between systems, the details of the specific system being used must be known before these attributes can be used.

Storage control attributes are associated with units of storage that are multiples of the page size. Each storage access is performed according to the storage control attributes of the specified storage location, as described below. The storage control attributes are the following.

- Write Through Required
- Caching Inhibited
- Memory Coherence Required
- Guarded
- []

These attributes have meaning only when an effective address is translated by the processor performing the storage access.

Programming Note

The Write Through Required and Caching Inhibited attributes are mutually exclusive because, as described below, the Write Through Required attribute permits the storage location to be in the data cache while the Caching Inhibited attribute does not.

Storage that is Write Through Required or Caching Inhibited is not intended to be used for general-purpose programming. For example, the ***lbarx***, ***lharx***, ***lwarx***, ***ldarx***, ***lqarx***, ***stbcx.***, ***sthcx.***, ***stwcx.***, ***stdcx.***, and ***stqcx***. instructions may cause the system data storage error handler to be invoked if they specify a location in storage having either of these attributes. To obtain the best performance across the widest range of implementations, storage that is Write Through Required or Caching Inhibited should be used only when the use of such storage meets specific functional or semantic needs or enables a performance optimization.

In the remainder of this section, "*Load* instruction" includes the *Cache Management* and other instructions that are stated in the instruction descriptions to be "treated as a *Load*" unless they are explicitly excluded, and similarly for "*Store* instruction".

1.6.1 Write Through Required

A store to a Write Through Required storage location is performed in main storage. A Store instruction that specifies a location in Write Through Required storage may cause additional locations in main storage to be accessed. If a copy of the block containing the specified location is retained in the data cache, the store is also performed in the data cache. The store does not cause the block to be considered to be modified in the data cache.

In general, accesses caused by separate *Store* instructions that specify locations in Write Through Required storage may be combined into one access. Such combining does not occur if the *Store* instructions are separated by a ***sync*** or ***eieio*** instruction.

1.6.2 Caching Inhibited

An access to a Caching Inhibited storage location is performed in main storage. A *Load* instruction that specifies a location in Caching Inhibited storage may cause additional locations in main storage to be accessed unless the specified location is also Guarded. An instruction fetch from Caching Inhibited storage may cause additional words in main storage to be accessed. No copy of the accessed locations is placed into the caches.

In general, non-overlapping accesses caused by separate *Load* instructions that specify locations in Caching Inhibited storage may be combined into one access, as may non-overlapping accesses caused by separate *Store* instructions that specify locations in Caching Inhibited storage. Such combining does not occur if the *Load* or *Store* instructions are separated by a ***sync*** instruction. Combining may also occur among such accesses from multiple processors that share a common memory interface. No combining occurs if the storage is also Guarded.

Programming Note

None of the memory barrier instructions prevent the combining of accesses from different processors. The Guarded storage attribute must be used in combination with Caching Inhibited to prevent such combining.

Prefixed instructions are not fetched from storage that is Caching Inhibited. If the instruction addressed by the current instruction address is a prefixed instruction and is in such storage, the system instruction storage error handler is invoked (see Section 7.5.5 of Book III).

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.

Engineering Note

Memory coherence can be implemented, for example, by an ownership protocol that allows at most one processor at a time to store to a given location in Memory Coherence Required storage.

A processor observing a storage access initiated by another processor or mechanism must honor the coherence requirements of that access, even if the observing processor last accessed the affected storage location as not Memory Coherence Required.

1.6.4 Guarded

A data access to a Guarded storage location is performed only if either (a) the access is caused by an instruction that is known to be required by the sequential execution model, or (b) the access is a load and the storage location is already in a cache. If the storage is also Caching Inhibited, only the storage location specified by the instruction is accessed; otherwise any storage location in the cache block containing the specified storage location may be accessed.

Except in ultravisor or hypervisor real addressing mode, instructions are not fetched from storage that is Guarded. Except in these addressing modes, if the instruction addressed by the current instruction address is in such storage, the system instruction storage error handler is invoked (see Section 7.5.5 of Book III).

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).

[]

1.7 Shared Storage

This architecture supports the sharing of storage between programs, between different instances of the same program, and between processors and other mechanisms. It also supports access to a storage location by one or more programs using different effective addresses. All these cases are considered storage sharing. Storage is shared in blocks that are an integral number of pages.

When the same storage location has different effective addresses, the addresses are said to be *aliases*. Each application can be granted separate access privileges to aliased pages.

1.7.1 Storage Access Ordering

The storage model for the ordering of storage accesses is weakly consistent. This model provides an opportunity for improved performance over a model that has stronger consistency rules, but places the responsibility on the program to ensure that ordering or synchronization instructions are properly placed when storage is shared by two or more programs.

[]

The order in which the processor performs storage accesses, the order in which those accesses are performed with respect to another processor or mechanism, and the order in which those accesses are performed in main storage may all be different. Several means of enforcing an ordering of storage accesses are provided to allow programs to share storage with other programs, or with mechanisms such as I/O devices. These means are listed below. The phrase “to the extent required by the associated Memory Coherence Required attributes” refers to the Memory Coherence Required attribute, if any, associated with each access.

- If two *Store* instructions or two *Load* instructions specify storage locations that are both Caching Inhibited and Guarded, the corresponding storage accesses are performed in program order with respect to any processor or mechanism.
- If a *Load* instruction depends on the value returned by a preceding *Load* instruction (because the value is used to compute the effective address specified by the second *Load*), the corresponding storage accesses are performed in program order with respect to any processor or mechanism to the extent required by the associated Memory Coherence Required attributes. This applies even if the dependency has no effect on program logic (e.g., the value returned by the first *Load* is ANDed with zero and then added to the effective address specified by the second *Load*).
- When a processor (P1) executes a *Synchronize* or ***eieio*** instruction a *memory barrier* is created, which orders applicable storage accesses pairwise, as follows. Let A be a set of storage accesses that includes all storage accesses associated with instructions preceding the barrier-creating instruction, and let B be a set of storage accesses that includes all storage accesses associated with instructions following the barrier-creating instruction. For each applicable pair a_i,b_j of storage accesses such that a_i is in A and b_j is in B, the memory barrier ensures that a_i will be performed with respect to any processor or mechanism, to the extent required by the associated Memory Coherence Required attributes, before b_j is performed with respect to that processor or mechanism.

  The ordering done by a memory barrier is said to be “cumulative” if it also orders storage accesses that are performed by processors and mechanisms other than P1, as follows.

  - A includes all applicable storage accesses by any such processor or mechanism that have been performed with respect to P1 before the memory barrier is created.
  - B includes all applicable storage accesses by any such processor or mechanism that are performed after a Load instruction executed by that processor or mechanism has returned the value stored by a store that is in B.

No ordering should be assumed among the storage accesses caused by a single instruction (i.e, by an instruction for which the access is not atomic), [] and no means are provided for controlling that order.

Programming Note

Because stores cannot be performed "out-of-order" (see Book III), if a *Store* instruction depends on the value returned by a preceding *Load* instruction (because the value returned by the *Load* is used to compute either the effective address specified by the *Store* or the value to be stored), the corresponding storage accesses are performed in program order. The same applies if *whether* the *Store* instruction is executed depends on a conditional *Branch* instruction that in turn depends on the value returned by a preceding *Load* instruction.

Because an ***isync*** instruction prevents the execution of instructions following the ***isync*** until instructions preceding the ***isync*** have completed, if an ***isync*** follows a conditional *Branch* instruction that depends on the value returned by a preceding *Load* instruction, the load on which the *Branch* depends is performed before any loads caused by instructions following the ***isync***. This applies even if the effects of the "dependency" are independent of the value loaded (e.g., the value is compared to itself and the *Branch* tests the EQ bit in the selected CR field), and even if the branch target is the sequentially next instruction.

With the exception of the cases described above and earlier in this section, data dependencies and control dependencies do not order storage accesses. Examples include the following.

- If a *Load* instruction specifies the same storage location as a preceding *Store* instruction and the location is in storage that is not Caching Inhibited, the load may be satisfied from a "store queue" (a buffer into which the processor places stored values before presenting them to the storage subsystem), and not be visible to other processors and mechanisms. A consequence is that if a subsequent *Store* depends on the value returned by the *Load*, the two stores need not be performed in program order with respect to other processors and mechanisms.
- Because a *Store Conditional* instruction may complete before its store has been performed, a conditional *Branch* instruction that depends on the CR0 value set by a *Store Conditional* instruction does not order the *Store Conditional*'s store with respect to storage accesses caused by instructions that follow the *Branch*.
- Because processors may predict branch target addresses and branch condition resolution, control dependencies (e.g., branches) do not order storage accesses except as described above. For example, when a subroutine returns to its caller the return address may be predicted, with the result that loads caused by instructions at or after the return address may be performed before the load that obtains the return address is performed.

Because processors may implement nonarchitected duplicates of architected resources (e.g., GPRs, CR fields, and the Link Register), resource dependencies (e.g., specification of the same target register for two Load instructions) do not order storage accesses.

Examples of correct uses of dependencies, ***sync*** and ***lwsync*** to order storage accesses can be found in B "Programming Examples for Sharing Storage " on page 1044.

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.

Programming Note

The first example below illustrates cumulative ordering of storage accesses preceding a memory barrier, and the second illustrates cumulative ordering of storage accesses following a memory barrier. Assume that locations X, Y, and Z initially contain the value 0.

Example 1:

Processor A:
stores the value 1 to location X

Processor B:
loads from location X obtaining the value 1, executes a ***sync*** instruction, then stores the value 2 to location Y

Processor C:
loads from location Y obtaining the value 2, executes a ***sync*** instruction, then loads from location X

Example 2:

Processor A:
stores the value 1 to location X, executes a ***sync*** instruction, then stores the value 2 to location Y

Processor B:
loops loading from location Y until the value 2 is obtained, then stores the value 3 to location Z

Processor C:
loads from location Z obtaining the value 3, executes a ***sync*** instruction, then loads from location X

In both cases, cumulative ordering dictates that the value loaded from location X by processor C is 1.

Engineering Note

It is permissible to perform a dependent load before the load on which it depends, if software accessing shared storage cannot tell the difference.

It is always permissible to prefetch a data cache block from non-Guarded storage based on predicting the effective address specified by a *Load* or *Store* instruction.

Engineering Note

Correct operation of the *Synchronize* and ***eieio*** instructions depends on both the processor and the storage subsystem.

The definition of memory barriers is not intended to preclude address pipelining. For example, if two applicable *Storage Access* instructions are separated by a *Synchronize* or ***eieio*** instruction, it is permissible for the data address associated with the second instruction to be presented to a given level of the storage hierarchy before the data access caused by the first instruction has completed at that level. However, if such pipelining is done, the processor must provide sufficient information that the storage subsystem can keep the storage accesses in the correct order, and the storage subsystem must do so.

Memory barriers need not order the following:

- the prefetching of cache blocks
- the casting out of cache blocks
- consistency operations initiated by other processors

1.7.1.1 Storage Ordering of Copy/Paste-Initiated Data Transfers

The Copy-Paste Facility (see Section 4.4) uses pairs of instructions to initiate 128-byte data transfers. They are referred to as “data transfers” to differentiate them from the “normal” storage accesses caused by or associated with loads, stores, and instructions that are treated as loads and stores. In the absence of barriers, the relative ordering among adjacent data transfers or data transfers and storage accesses is not defined, and the sequential execution model and coherence-required ordering relationships do not apply. To establish order between adjacent data transfers or between data transfers and storage accesses, ***hwsync*** must be used. See the description of the *Synchronize* instruction in Section 4.6.3 for more information.

Programming Note

It may be helpful to think of a ***copy***/***paste.*** pair sending the real storage addresses of the 128-byte source and destination to an asynchronous data transfer engine completely separate from the processor that is executing the ***copy*** and ***paste.*** instructions. The data transfers collect in the engine’s queue. The engine may perform the data transfers in any order, and with the only relative timing relationship to adjacent transfers and accesses being determined by ***hwsync***.

1.7.1.2 Storage Ordering of Stores to Persistent Storage

A location in a region of main storage that is backed by persistent storage is considered to be modified relative

to persistent storage if the location has been modified in main storage (e.g., by a *Store* instruction) and the modified data have not been written to persistent storage. A store has updated persistent storage when a load by any given processor, from the location accessed by the store, would return the value stored (or a value stored subsequently) if system power were lost temporarily between the time the store has updated persistent storage and the time the load is performed. A store may update persistent storage significantly later than it updates main storage. The ***dcbstps*** (data cache block store to persistent storage) and ***dcbfps*** (data cache block flush to persistent storage) instructions can be used to write modified locations in a block to persistent storage (and to perform the functions of *Data Cache Block Store* and *Data Cache Block Flush* respectively). The ***phwsync*** (persistent heavyweight sync) and ***plwsync*** (persistent lightweight sync) instructions can be used to establish order for these writes to persistent storage. A store may update persistent storage even in the absence of ***dcbstps*** and ***dcbfps*** instructions targetting the cache block(s) affected by the store. See Section 4.3.2 and Section 4.6.3 for more information.

Except in this section and in other material dealing with persistent storage, references to the storage subsystem in this document assume system power is not lost unless otherwise stated or obvious from context.

Programming Note

On the POWER9 processor, ***dcbf*** performs the functions required of ***dcbfps*** and ***dcbstps***. The encodings chosen for ***dcbfps*** and ***dcbstps*** are decoded as ***dcbf*** on the POWER9 processor.

On the POWER9 processor, ***lwsync*** and ***hwsync*** perform the functions required of ***plwsync*** and ***phwsync***, respectively. The encodings chosen for ***plwsync*** and ***phwsync*** are decoded as ***lwsync*** and ***hwsync***, respectively, on the POWER9 processor.

The encodings for ***dcbfps***, ***dcbstps***, ***plwsync***, and ***phwsync*** were chosen to enable software developed to control updates to persistent storage on processors that comply with Version 3.1 and subsequent versions of the architecture to run compatibly on the POWER9 processor.

Programming Note

The ordering of loads from persistent storage is not discussed because loads are satisfied from the nearest source (store buffer, nearest cache, or memory) of a consistent value for a datum.

1.7.1.3 Storage Ordering of I/O Accesses

A “coherence domain” consists of all processors and all interfaces to main storage. Memory reads and writes initiated by mechanisms outside the coherence domain are performed within the coherence domain in the order in which they enter the coherence domain and are performed as coherent accesses.

Engineering Note

In some cases, the I/O interface to coherent memory may reorder accesses initiated by different I/O adapters. Designers should be aware of the requirements and behavior of I/O interfaces.

1.7.2 Atomic Update

The *Load And Reserve* and *Store Conditional* instructions together permit atomic update of a shared storage location. There are byte, halfword, word, doubleword, and quadword forms of each of these instructions. Described here is the operation of the word forms ***lwarx*** and ***stwcx.***; operation of the byte, halfword, doubleword, and quadword forms ***lbarx***, ***stbcx.***, ***lharx***, ***sthcx.***, ***ldarx***, ***stdcx.***, ***lqarx***, and ***stqcx.*** 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 implementation. The remainder of this paragraph assumes that these two conditions are satisfied. If the storage locations specified by the two instructions differ, or if a *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 B “Programming Examples for Sharing Storage ” on page 1044.

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.

Programming Note

If a virtual address is reassigned to a different real page, a reservation established at the virtual address before the reassignment will not be cleared by a store to the new real page by some other processor or mechanism. (As described in Section 1.7.2.1, reservations are held on real addresses.) If *Store Conditional* instructions did not suppress the store when the storage location specified by the *Store Conditional* instruction is in a different real page from the storage location specified by the corresponding *Load And Reserve* instruction, such virtual address reassignment could permit a *Store Conditional* instruction that specifies the same virtual address as the corresponding *Load And Reserve* instruction, and logically should fail because the other processor or mechanism stored to the virtual address, to succeed.

This real address checking cannot detect that the virtual page in which the reservation was established has been moved to a new real page and back again to the original real page that was accessed by the *Load And Reserve* instruction. It also cannot detect that the real address of the storage location specified by a *Store Conditional* instruction is the same as the real address of the reservation, or is in the same real page as the reservation, *only* because the virtual page containing the storage location specified by the *Store Conditional* instruction has been moved to the real page that was accessed by the corresponding *Load And Reserve* instruction. Privileged software that moves a virtual page should clear the reservation on the processor it is running on in order to ensure that a *Store Conditional* instruction executed by that processor does not succeed in these cases. (If the software that moves the virtual page uses *Load And Reserve* and *Store Conditional* for its own purposes, the clearing of the original reservation will happen naturally. The stores that occur naturally as part of moving the virtual page will cause any reservations, held by other processors, in the target real page to be cleared.)

1.7.2.1 Reservations

The ability to emulate an atomic operation using ***lwarx*** and ***stwcx.*** is based on the conditional behavior of ***stwcx.***, the reservation created by ***lwarx***, and the clearing of that reservation if the target storage location is modified by another processor or mechanism before the ***stwcx.*** performs its store.

A reservation is held on an aligned unit of real storage called a reservation granule. The size of the reservation granule is 2^n bytes, where n is implementation-dependent but is always at least 4 (thus the minimum reservation granule size is a quadword), and where 2^n is not larger than the smallest real page size supported by the implementation. The reservation granule associated with effective address EA contains the real address to which EA maps. ("real_addr(EA)" in the RTL for the *Load And Reserve* and *Store Conditional* instructions stands for "real address to which EA maps".) The reservation also has an associated length, which is equal to the storage operand length, in bytes, of the *Load And Reserve* instruction that established the reservation.

A processor has at most one reservation at any time. A reservation is established by executing a ***lbarx***, ***lharx***, ***lwarx***, ***ldarx***, or ***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 1-8 apply only if the relevant access is performed. (For example, an access that would ordinarily be caused by an instruction might not be performed if the instruction causes the system error handler to be invoked.)

1. The processor holding the reservation executes another ***lbarx***, ***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. Some other processor executes a *Store or* ***dcbz*** that specifies a location in the same reservation granule.
5. Some other processor executes a ***dcbtst***, or ***dcbt*** that specifies a location in the same reservation granule: whether the reservation is lost is undefined. (For a ***dcbtst*** instruction that specifies a data stream, "location" in the preceding sentence includes all locations in the data stream.)
6. Any processor modifies a Reference or Change bit in the same reservation granule: the reservation is lost if the modification is atomic; otherwise

whether the reservation is lost is undefined. (See Section 6.7.12 of Book III)

Engineering Note

Atomic Reference and Change bit modifications by hardware must cause reservation loss in order to interact correctly with software that emulates the atomic modification using *Load And Reserve* and *Store Conditional* instructions.

7. Some mechanism other than a processor modifies a storage location in the same reservation granule.
8. An interrupt (see Book III) occurs on the processor holding the reservation: the interrupt itself does not clear the reservation, but system software invoked by the interrupt may clear the reservation.
9. Implementation-specific characteristics of the coherence mechanism cause the reservation to be lost.

Engineering Note

Such reservation loss should not impede forward progress (see Section 1.7.2.2).

The reservation is also used by the ***waitrsv*** instruction (see Section 4.6.4).

Architecture Note

In the preceding list, all instructions that can cause reservation loss on other processors require write authority. This is necessary because an instruction that did not require write authority and caused reservation loss on other processors could be used to implement a "covert channel". Any new instructions that are added to the list in the future as being permitted to cause reservation loss on other processors must require write authority.

Virtualized Implementation Note

A reservation may be lost if:

- Software executes a privileged instruction or utilizes a privileged facility
- Software accesses storage not intended for general-purpose programming

Programming Note

One use of ***lwarx*** and ***stwcx.*** is to emulate a "Compare and Swap" primitive like that provided by the IBM System/370 Compare and Swap instruction; see Section B.1, "Atomic Update Primitives" on page 1044. A System/370-style Compare and Swap checks only that the old and current values of the word being tested are equal, with the result that programs that use such a Compare and Swap to control a shared resource can err if the word has been modified and the old value subsequently restored. The combination of ***lwarx*** and ***stwcx.*** improves on such a Compare and Swap, because the reservation reliably binds the ***lwarx*** and ***stwcx.*** together. The reservation is always lost if the word is modified by another processor or mechanism between the ***lwarx*** and ***stwcx.***, so the ***stwcx.*** never succeeds unless the word has not been stored into (by another processor or mechanism) since the ***lwarx***.

Programming Note

In general, programming conventions must ensure that ***lwarx*** and ***stwcx.*** specify addresses that match; a ***stwcx.*** should be paired with a specific ***lwarx*** to the same storage location. Situations in which a ***stwcx.*** may erroneously be issued after some ***lwarx*** other than that with which it is intended to be paired must be scrupulously avoided. For example, there must not be a context switch in which the processor holds a reservation in behalf of the old context, and the new context resumes after a ***lwarx*** and before the paired ***stwcx.***. The ***stwcx.*** in the new context might succeed, which is not what was intended by the programmer. Such a situation must be prevented by executing a ***stbcx.***, ***sthcx***., ***stwcx.***, ***stdcx.***, or ***stqcx.*** that specifies a dummy writable aligned location as part of the context switch; see Section 7.4.3 of Book III.

Programming Note

Because the reservation is lost if another processor stores anywhere in the reservation granule, lock words (or bytes, halfwords, or doublewords) should be allocated such that few such stores occur, other than perhaps to the lock word itself. (Stores by other processors to the lock word result from contention for the lock, and are an expected consequence of using locks to control access to shared storage; stores to other locations in the reservation granule can cause needless reservation loss.) Such allocation can most easily be accomplished by allocating an entire reservation granule for the lock and wasting all but one word. Because reservation granule size is implementation-dependent, portable code must do such allocation dynamically.

Similar considerations apply to other data that are shared directly using ***lwarx*** and ***stwcx.*** (e.g., pointers in certain linked lists; see Section B.3, "List Insertion" on page 1048).

Engineering Note

lwarx, ***stwcx.***, and ***waitrsv*** have a data dependence on the processor reservation resource.

Engineering Note

If memory coherence is supported, reservations must take part in the coherence mechanism. A reservation must be cleared if another processor or mechanism receives authorization from the coherence mechanism to store to the reservation granule.

Implementations must continue to hold a reservation when the cache block containing the reservation granule (here called the "reserved block") is evicted. The reservation must continue to participate in the coherence protocol. In a snooping implementation, it must join in snooping. In a directory-based implementation, it must register its interest in the reserved block with the directory (shared-read access). The alternative approach of holding the reserved block in the cache until it is explicitly invalidated (e.g., by another processor's store or ***dcbf***) or the *Store Conditional* instruction completes, which was permitted in versions of the architecture that precede Version 2.06, is no longer possible because, starting with V. 2.06, ***dcbf*** is not allowed to cause reservation loss.

1.7.2.2 Forward Progress

Forward progress in loops that use ***lwarx*** and ***stwcx.*** is achieved by a cooperative effort among hardware, system software, and application software.

The architecture guarantees that when a processor executes a ***lwarx*** to obtain a reservation for location X and then a ***stwcx.*** to store a value to location X, either

1. the ***stwcx.*** succeeds and the value is written to location X, or
2. the ***stwcx.*** fails because some other processor or mechanism modified location X, or
3. the ***stwcx.*** fails because the processor's reservation was lost for some other reason.

In Cases 1 and 2, the system as a whole makes progress in the sense that some processor successfully modifies location X. Case 3 covers reservation loss required for correct operation of the rest of the system. This includes cancellation caused by some other processor or mechanism writing elsewhere in the reservation granule, cancellation caused by the operating system in managing certain limited resources such as real storage, and cancellation caused by any of the other effects listed in see Section 1.7.2.1.

An implementation may make a forward progress guarantee, defining the conditions under which the system as a whole makes progress. Such a guarantee must specify the possible causes of reservation loss in Case 3. While the architecture alone cannot provide such a guarantee, the characteristics listed in Cases 1 and 2 are necessary conditions for any forward progress guarantee. An implementation and operating system can build on them to provide such a guarantee.

Virtualized Implementation Note

On a virtualized implementation, Case 3 includes reservation loss caused by the virtualization software. Thus, on a virtualized implementation, a reservation may be lost at any time without apparent cause. The virtualization software participates in any forward progress assurances, as described above.

Programming Note

The architecture does not include a "fairness guarantee". In competing for a reservation, two processors can indefinitely lock out a third.

[]

1.8 Instruction Storage

The instruction execution properties and requirements described in this section, including its subsections, apply only to instruction execution that is required by the sequential execution model.

Engineering Note

Any violation of these properties and requirements due to out-of-order execution of instructions must not be visible to software.

In this section, including its subsections, it is assumed that all instructions for which execution is attempted are

in storage that is not Caching Inhibited and (unless instruction address translation is disabled; see Book III) is not Guarded, and from which instruction fetching does not cause the system error handler to be invoked (e.g., from which instruction fetching is not prohibited by the "address translation mechanism" or the "storage protection mechanism"; see Book III).

Programming Note

The results of attempting to execute instructions from storage that does not satisfy this assumption are described in Section 1.6.2 and Section 1.6.4 of this Book and in Book III.

[]

The instruction cache is not necessarily kept consistent with the data cache or with main storage. It is the responsibility of software to ensure that instruction storage is consistent with data storage when such consistency is required for program correctness.

Architecture Note

The Power ISA does not acknowledge the existence of temporary, non-stateful instruction storage such as instruction prefetch line buffers, instruction pre-decode buffers, and the like. Their contents, prefetched instructions, are considered to be among the instructions that are in the process of being executed. However, the architecture places no limitations on reuse of their contents to repeatedly satisfy requests by the instruction fetching mechanism, and in so doing, creates a sort of invisible cache. (***icbi*** has no effect on the contents of these buffers.) One function of the context synchronization operation is to invalidate the contents of this invisible cache so that subsequent instructions are fetched and processed in the new context. The additional step of making the consistency visible to the instruction fetching mechanism is why an ***isync*** instruction is always included in the sequence of instructions that make instruction storage consistent with data storage. So in a statement such as the one preceding this Architecture Note, the expectation is that in addition to maintaining cache consistency using ***icbi***, software will also cause context synchronization where appropriate. The Programming Note below provides examples.

After one or more bytes of a storage location have been modified and before an instruction located in that storage location is executed, software must execute the appropriate sequence of instructions to make instruction storage consistent with data storage. Otherwise the result of attempting to execute the instruction is boundedly undefined except as described in Section 1.8.1, "Concurrent Modification and Execution of Instructions" on page 983.

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 of more bytes of that location such that, in data storage, the location contains the instruction A1; and location X is wholly contained in a single cache block. The following instruction sequence will make instruction storage consistent with data storage such that if the ***isync*** was in location X-4, the instruction A1 in location X would be executed immediately after the ***isync***.

```
dcbst   X     #copy the block to main storage
sync          #order copy before invalidation
icbi    X     #invalidate copy in instr cache
isync         #discard prefetched instructions
```

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
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
```

In the preceding instruction sequence any context synchronizing instruction (e.g., ***rfid***) can be used instead of ***isync***. (For Case 1 only ***isync*** can be used.)

For both cases, if two or more instructions in separate data cache blocks have been modified, the ***dcbst*** instruction in the examples must be replaced by a sequence of ***dcbst*** instructions such that each block containing the modified instructions is copied back to main storage. Similarly, for ***icbi*** the sequence must invalidate each instruction cache block containing a location of an instruction that was modified. The ***sync*** instruction that appears above between "***dcbst*** X" and "***icbi*** X" would be placed between the sequence of ***dcbst*** instructions and the sequence of ***icbi*** instructions.

1.8.1 Concurrent Modification and Execution of Instructions

The phrase "concurrent modification and execution of instructions" (CMODX) refers to the case in which a processor fetches and executes an instruction from instruction storage which has not been made consistent with data storage. This section describes the only case in which executing this instruction under these conditions produces defined results.

Programming Note

The architecture does not support CMODX modification of an entire prefixed instruction because it may be unaligned and therefore impossible to modify atomically. Furthermore, no need has arisen to motivate the creation of special cases where it must work. The architecture also does not support CMODX modification of the suffix of a prefixed instruction.

In the remainder of this section the following terminology is used.

- Location X is an arbitrary four-byte word-aligned storage location.
- X_0 is the value of the contents of location X for which software has made the location X in instruction storage consistent with data storage.
- X_1, X_2, ..., X_n are the sequence of the first n values occupying location X after X_0.
- X_n is the first value of X subsequent to X_0 for which software has again made instruction storage consistent with data storage.
- The "patch class" of words consists of the following.
  - the I-form *Branch* instruction (***b***[***l***][***a***])
  - the preferred no-op instruction (***ori*** 0,0,0)
  - the prefix of the Prefixed No-op instruction (***pnop***)
  - the D-form and X-form *Trap* instructions for which TO = 0b11111
- The "instruction from location X" includes both the case of a word instruction contained in location X and the case of a prefixed instruction for which the prefix is contained in location X.

If the instruction from location X is executed after the copy of location X in instruction storage is made consistent for the value X_0 and before it is made consistent for the value X_n, the results of executing the instruction are defined if and only if the following conditions are satisfied.

1. The stores that place the values X_1, ..., X_n into location X are atomic stores that modify all four bytes of location X.
2. The sequence of X_i values is one of two types:
   - any sequence in which each X_i is a patch class word, or
   - any sequence that is comprised of exactly two unique word values, at least one of which is a patch class word.
3. If a sequence of X_i values contains a prefix, the only word instructions the sequence can contain are the patch class *Branch* and patch class *Trap* instructions. In this use, the target of the *Branch* instruction must not be the word following the *Branch* instruction.
4. Location X is in storage that is Memory Coherence Required.

Programming Note

If the instruction from location X is a prefixed instruction, it should obey the rules for prefixed instructions. In particular:

- The instruction should not cross a 64-byte boundary. (Thus if the EA of location X is equal to 60 modulo 64, none of the X_i should be prefixes.)
- If the instruction is ***pnop***, the word at location X+4 should not be a *Branch* instruction, ***rfebb***, a context synchronizing instruction other than ***isync***, or a "Service Processor Attention" instruction.

If these conditions are satisfied, [] each execution of an instruction from location X will use some X_i, $0 \leq i \leq n$. The value of the ordinate i associated with each value executed may be different and the sequence of ordinates i associated with a sequence of values executed is not constrained, (e.g., a valid sequence of executions of the instruction from location X could use the sequence X_i, X_{i+2}, then X_{i-1}). If these conditions are not satisfied, the results of each such execution of an instruction from location X are boundedly undefined, and may include causing inconsistent information to be presented to the system error handler.

Programming Note

An example of how failure to satisfy the requirements given above can cause inconsistent information to be presented to the system error handler is as follows. If the value X_0 (an illegal instruction) is executed, causing the system illegal instruction handler to be invoked, and before the error handler can load X_0 into a register, X_0 is replaced with X_1, an *Add Immediate* instruction, it will appear that a legal instruction caused an illegal instruction exception.

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 1.8 where one program is creating instructions to be executed by one or more other programs.

In place of the *Store* to a flag to indicate to the other programs that the code is ready to be executed, the program that is applying the patch would replace an instruction in the original program with an I-form *Branch* instruction that would cause any program executing the *Branch* to branch to the newly created code. The first instruction in the newly created code must be an ***isync***, which will cause any prefetched instructions to be discarded, ensuring that the execution is consistent with the newly created code. The instruction storage location containing the ***isync*** instruction in the patch area must be consistent with data storage with respect to the processor that will execute the patched code before the *Store* which stores the new *Branch* instruction is performed.

Programming Note

The ability to modify instructions that may be being executed provides opportunities for significant performance benefits in interpretive environments (e.g., Java), and can be used to instrument code dynamically to isolate critical hardware and software bugs.

Engineering Note

Architectural support for CMODX implies a commitment on the part of hardware that unsynchronized updates to location X will eventually result in the execution of different values of X_i. In practice this means there must be a limitation on the reuse of data in the I-cache and instruction buffers (or an awareness of updates to instruction storage).

Engineering Note

Supporting CMODX in processor designs which execute an instruction multiple times, using state from previous executions to determine how a subsequent one will be performed, has been difficult in some cases. Similarly, processor designs in which an instruction may be fetched multiple times for various phases of its execution may fetch a different instruction on any fetch of the instruction, leading to a potential paradox. When considering such an approach to a processor design, the designer should consider the need to support CMODX, the limitations on boundedly undefined results, and the complexity of verifying that these requirements are met.

Chapter 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 7.6 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.

Programming Note

There are many events that might cause a *Load* or *Store* instruction to be restarted. For example, a hardware error may cause execution of the instruction to be aborted after part of the access has been performed, and the recovery operation could then cause the aborted instruction to be re-executed.

When an instruction is aborted after being partially executed, the contents of the instruction pointer indicate that the instruction has not been executed, however, the contents of some registers may have been altered and some bytes within the storage operand may have been accessed. The following are examples of an instruction being partially executed and altering the program state even though it appears that the instruction has not been executed.

1. *Load Multiple*, *Load String*: Some registers in the range of registers to be loaded may have been altered.
2. Any *Store* instruction, ***dcbz***: Some bytes of the storage operand may have been altered.

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 3.1.

Programming Note

The ability to access the low-order half of the PPR (and thus the use of ***mfppr*** and ***mtppr***) might be phased out in a future version of the architecture.

Architecture Note

It may be desirable, in the future, to define several bits immediately above and/or immediately below the PRI field (e.g., $PPR32_{40:42}$, $PPR_{14:16}$) to be a priority extension field. Therefore effort should be made to avoid assigning another meaning to these bits.

Programming Note

By setting the PRI field, a programmer may be able to improve system throughput by causing system resources to be used more efficiently.

E.g., if a program is waiting on a lock (see Section B.2), it could set low priority, with the result that more processor resources would be diverted to the program that holds the lock. This diversion of resources may enable the lock-holding program to complete the operation under the lock more quickly, and then relinquish the lock to the waiting program.

Programming Note

or Rx,Rx,Rx can be used to modify the PRI field; see Section 3.2.

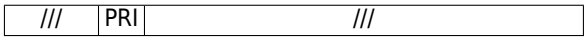
Programming Note

When the system error handler is invoked, the PRI field may be set to an undefined value.

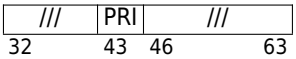
Programming Note

Additional priority levels are defined in Book III.

PPR:



PPR32:



Bit(s)	Definition
11:13	**Program Priority** (PRI) ***($PPR32_{43:45}$)***

001	very low
010	low
011	medium low
100	medium
101	medium high

Programs can always set the PRI field to very low, low, medium low, and medium priorities; programs may be allowed to set the PRI field to medium high priority during certain time intervals. (See Section 5.3.6.) If the program priority is medium high when the time interval expires or if an attempt is made to set the priority to medium high when it is not allowed, the PRI field is set to medium.

If other values are written to this field, the PRI field is not changed. (See Section 5.3.5 of Book III for additional information.)

All other fields are reserved.

Figure 3.1: Program Priority Register

3.2 “or” Instruction

Setting the PPR

The ***or*** *Rx,Rx,Rx* (see Book I) instruction can be used to set PPR_{PRI} as shown in Table 3.1. ***or.*** *Rx,Rx,Rx* does not set PPR_{PRI}.

Rx	PPR_{PRI}	Priority
31	001	very low
1	010	low
6	011	medium low
2	100	medium
5	101	medium high

Table 3.1: Priority levels for ***or*** *Rx,Rx,Rx*

Programs can always set the PRI field to very low, low, medium low, and medium priorities; programs may be allowed to set the PRI field to medium high priority during certain time intervals. (See Section 5.3.6 of Book III.) If the program priority is medium high when the time interval expires or if an attempt is made to set the priority to medium high when it is not allowed, the PRI field is set to medium.

Programming Note

Warning: Other forms of ***or*** *Rx,Rx,Rx* that are not described in this section and in Section 4.3.3 may also cause program priority to change. Use of these forms should be avoided except when software explicitly intends to alter program priority. If a no-op is needed, the preferred no-op (***ori*** *0,0,0*) should be used.

Chapter 4. Storage Control Instructions

4.1 Parameters Useful to Application Programs

It is suggested that the operating system provide a service that allows an application program to obtain the following information.

1. The virtual page sizes
2. Coherence block size
3. Reservation granule size
4. An indication of the cache model implemented (e.g., Harvard-style cache, combined cache)
5. Instruction cache size
6. Data cache size
7. Instruction cache block size
8. Data cache block size
9. Instruction cache associativity
10. Data cache associativity
11. Number of stream IDs supported for the stream variant of ***dcbt***
12. Factors for converting the Time Base to seconds

If the caches are combined, the same value should be given for an instruction cache attribute and the corresponding data cache attribute.

Architecture Note

All processors in a symmetric multiprocessor must be identical with respect to the cache model, the coherence block size, and the reservation granule size.

4.2 Data Stream Control Register (DSCR)

The layout of the Data Stream Control Register (DSCR) is shown in Figure 4.1 below.

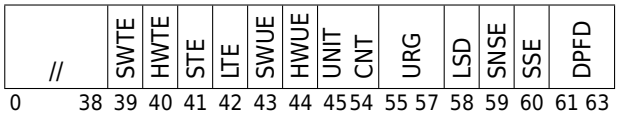


Figure 4.1: Data Stream Control Register

Bit(s) Definition

39 **Software Transient Enable** (SWTE)
- 0 SWTE is disabled.
- 1 Applies the transient attribute to software-defined streams.

40 **Hardware Transient Enable** (HWTE)
- 0 HWTE is disabled.
- 1 Applies the transient attribute to hardware-detected streams.

41 **Store Transient Enable** (STE)
- 0 STE is disabled.
- 1 Applies the transient attribute to store streams.

42 **Load Transient Enable** (LTE)
- 0 LTE is disabled.
- 1 Applies the transient attribute to load streams.

43 **Software Unit count Enable** (SWUE)
- 0 SWUE is disabled.
- 1 Applies the unit count to software-defined streams.

44 **Hardware Unit count Enable** (HWUE)
- 0 HWUE is disabled.
- 1 Applies the unit count to hardware-detected streams.

45:54 **Unit Count** (UNITCNT)

Number of units in data stream.

55:57 **Depth Attainment Urgency** (URG)

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

58 **Load Stream Disable** (LSD)
- 0 No effect.
- 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 992, for information on streams and how software may specify them.

Programming Note

The URG, LSD, SNSE and SSE fields do not affect the initiation of streams specified using the ***dcbt*** and ***dcbtst*** instructions.

Note that even when SNSE is not set, hardware may detect Stride-N streams in intervals when they access elements that map to sequential cache blocks.

Programming Note

In order for the DSCR to apply the transient attribute to streams, at least two of the four enable bits must be set: one to choose a type of access (load or store), and one to choose a kind of prefetching (software-defined or hardware-detected).

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.

Engineering Note

Warning: Hardware prefetching and the speculative translation supporting it must be limited to the context of the software that is executing at the time. Failure to apply this limitation may produce incorrect results because privileged software may track the threads on which a given context executes and limit translation cache invalidation based on that tracking. It may also cause hardware paradoxes, for example in the form of multiple hits in translation caches. In addition to being sensitive to explicit context such as the state of LPIDR, PIDR, and $MSR_{IR\ DR}$, the prefetching mechanism must also be aware of implied context, e.g. by avoiding prefetching across quadrant boundaries when HR=1. See Section 6.9.3 of Book III for additional guidance.

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 968, storage protection, reference and change recording, the storage access ordering described in Section 1.7.1, and Performance Monitor events (see Section 11.4.5 of Book III).

[]

Some *Cache Management* instructions contain a CT field that is used to specify a cache level within a cache hierarchy or a portion of a cache structure to which the instruction is to be applied. The correspondence between the CT value specified and the cache level is shown below.

CT Field Value	Cache Level
0	Primary Cache
2	Secondary Cache

CT values not shown above may be used to specify implementation-dependent cache levels or implementation-dependent portions of a cache structure.

Engineering Note

If the applicable cache does not exist, all of the *Cache Management* instructions except ***dcbz*** are treated as no-ops. If the data cache does not exist, ***dcbz*** must either (a) set to zero all bytes of the area of main storage that corresponds to the specified block, or (b) cause an Alignment interrupt to be taken.

If, at any level of the storage hierarchy, a combined cache is implemented such that locations in that cache lack an indication of whether they were fetched as data or as instructions, the locations must be treated as if they were fetched as data and must not be treated as if they were fetched as instructions. E.g., ***dcbf*** or ***dcbfl*** must flush and invalidate them, and ***icbi*** must not invalidate them unless any modified data is first written to storage. (Permitting ***icbi*** to invalidate a block that was fetched as data would permit it to invalidate modified data, creating a security and data integrity exposure.)

Engineering Note

An example of the requirements of the sequential execution model with respect to *Cache Management* instructions is that a *Load* instruction that specifies a storage location in the block specified by a preceding ***dcbf*** or ***dcbfl*** instruction must be satisfied from main storage (if the location is in storage that is not Memory Coherence Required) or from coherent storage (if the location is in storage that is Memory Coherence Required), and not from the copy of the location that existed in the data cache when the ***dcbf*** or ***dcbfl*** instruction was executed.

Cache reload buffers must be treated as if they were part of the corresponding cache. Cache reload requests must also be considered with respect to the execution of a *Cache Management* instruction specifying a block for which a reload request is outstanding. For example, if a cache reload request for a given data cache block is outstanding when a ***dcbf*** or ***dcbfl*** instruction is executed specifying the same block, the contents returned by the reload request must not be used to satisfy a subsequent Load instruction. Similarly, if a cache reload request for a given instruction cache block is pending when an ***icbi*** instruction is executed specifying the same block, the contents returned by the reload request must not be used to satisfy an instruction fetch for instructions following a subsequent ***isync*** instruction.

An example of the requirements of data dependencies with respect to *Cache Management* instructions is that if a ***dcbf*** or ***dcbfl*** instruction depends on the value returned by a preceding Load instruction, the invalidation caused by the ***dcbf*** or ***dcbfl*** must be performed after the load has been performed.

4.3.1 Instruction Cache Instructions

Instruction Cache Block Invalidate X-form

icbi RA,RB

31	///	RA	RB	982	/
0	6	11	16	21	31

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

If the block containing the byte addressed by EA is in storage that is Memory Coherence Required and a block containing the byte addressed by EA is in the instruction cache of any processors, the block is invalidated in those instruction caches.

If the block containing the byte addressed by EA is in storage that is not Memory Coherence Required and the block is in the instruction cache of this processor, the block is invalidated in that instruction cache.

The function of this instruction is independent of whether the block containing the byte addressed by EA is in storage that is Write Through Required or Caching Inhibited.

This instruction is treated as a *Load* (see Section 4.3), except that reference and change recording need not be done.

Special Registers Altered:

None

Programming Note

Because the instruction is treated as a *Load*, the effective address is translated using translation resources that are used for data accesses, even though the block being invalidated was copied into the instruction cache based on translation resources used for instruction fetches (see Book III).

Programming Note

The invalidation of the specified block need not have been performed with respect to the processor executing the ***icbi*** instruction until a subsequent ***isync*** instruction has been executed by that processor. No other instruction or event has the corresponding effect.

Instruction Cache Block Touch X-form

icbt CT,RA,RB

31	/	CT	RA	RB	22	/
0	6	7	11	16	21	31

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

Engineering Note

The CT field could be used to permit the programmer to specify the level of the cache hierarchy into which the addressed instructions should be fetched or the portion (e.g., way) of the cache to be used.

4.3.2 Data Cache Instructions

The Data Cache instructions control various aspects of the data cache.

TH field in the *dcbt* and *dcbtst* instructions

Described below are the TH field values for the ***dcbt*** and ***dcbtst*** instructions. For all TH field values which are not listed, the hint provided by the instruction is undefined.

TH=0b00000

If TH=0b00000, the ***dcbt***/***dcbtst*** instruction provides a hint that the program will probably soon access the block containing the byte addressed by EA.

TH=0b01000 – 0b01111

The ***dcbt***/***dcbtst*** instructions provide hints regarding a sequence of accesses to data elements, or indicate the expected use thereof. Such a sequence is called a "data stream", and a ***dcbt***/***dcbtst*** instruction in which TH is set to one of these values is said to be a "data stream variant" of ***dcbt***/***dcbtst***. In the remainder of this section, "data stream" may be abbreviated to "stream".

A data stream to which a program may perform *Load* accesses is said to be a "load data stream", and is described using the data stream variants of the ***dcbt*** instruction. A data stream to which a program may perform *Store* accesses is said to be a "store data stream", and is described using the data stream variants of the ***dcbtst*** instruction.

When, and how often, effective addresses for a data stream are translated is implementation-dependent.

Each data element is associated with a *unit* of storage, which is the aligned 128-byte location in storage that contains the first byte of the element. The data stream variants may be used to specify the address of the beginning of the data stream, the displacement (stride) between the first byte of successive elements, and the number of unique units of storage that are associated with all of the data elements. If the stride is specified, both the stride and the address of the first element are specified at 4 byte granularity. If the stride is not specified, the address of the first element is the address of the first unit.

Programming Note

The architecture does not provide a way to specify the size of the data elements that compose a stream. An implementation may assume some fixed size for all data elements. As a result, depending on the offset, stride, and size (and in particular whether the elements are aligned), the implementation may reduce the latency for accessing only a portion of some of the elements. A future version of the architecture may enable the specification of element size to avoid this limitation.

Each such data stream is associated, by software, with a stream ID, which is a resource that the processor uses to distinguish the data stream from other such data streams. The number of stream IDs is an implementation-dependent value in the range 1:16. Stream IDs are numbered sequentially starting from 0.

Programming Note

The number of stream IDs available for a program to use may be dependent on the current degree of multithreading of a processor in the system.

The encodings of the TH field and of the corresponding EA values are as follows. In the EA layout diagrams, fields shown as "/"s are reserved. These reserved fields are treated in the same manner as the corresponding case for instruction fields (see Section 1.3.3 of Book I). If a reserved value is specified for a defined EA field, or if a TH value is specified that is not explicitly defined below, the hint provided by the instruction is undefined.

TH **Description**

01000 The ***dcbt***/***dcbtst*** instruction provides a hint that describes certain attributes of a data stream, and may indicate that the program will probably soon access the stream.

The EA is interpreted as follows.

EATRUNC	D	UG	/	ID
0	57		59	60 63

Bit(s) Definition

0:56 **EATRUNC**

High-order 57 bits of the effective address of the first element of the data stream. (i.e., the effective address of the first unit of the stream is EATRUNC || $^{7}0$)

57 **Direction** (D)

0 Subsequent elements have increasing addresses.

1 Subsequent elements have decreasing addresses.

58 **Unlimited/GO** (UG)

0 No information is provided by the UG field.

1 The number of elements in the data stream is unlimited, the elements are adjacent to each other, the program's need for each element of the stream is not likely to be transient, and the program will probably soon access the stream.

59 Reserved

60:63 **Stream ID** (ID)

Stream ID to use for this data stream.

01010 The ***dcbt***/***dcbtst*** instruction provides a hint that describes certain attributes of a data stream, or indicates that the program will probably soon access data streams that have been

described using data stream variants of the ***dcbt***/***dcbtst*** instruction, or will probably no longer access such data streams.

The EA is interpreted as follows. If GO=1 and S≠0b00 the hint provided by the instruction is undefined; the remainder of this instruction description assumes that this combination is not used.

///	GO	S	/	DEP	//	UNITCNT	T	U	/	ID
0	32		35	36	39	47	57		59	60 63

Bit(s) Definition

0:31 Reserved

32 **GO**

- 0 No information is provided by the GO field.
- 1 For ***dcbt***, the program will probably soon access all nascent load and store data streams that have been completely described, and will probably no longer access all other nascent load and store data streams. All other fields of the EA are ignored. ("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.)
- 11 For ***dcbt***, 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 = $DSCR_{DPFD}$
- 1 none
- 2 shallowest
- 3 shallow
- 4 medium
- 5 deep
- 6 deeper
- 7 deepest

39:46 Reserved

47:56 **Unit Count** (UNITCNT)

Number of units in data stream.

57 **Transient** (T)

If T=1, the program's need for each element of the data stream is likely to be transient (i.e., the time interval during which the program accesses the element is likely to be short).

58 **Unlimited** (U)

If U=1, the number of units in the data stream is unlimited (and the UNITCNT field is ignored).

59 Reserved

60:63 **Stream ID** (ID)

Stream ID to use for this data stream (GO=0 and S=0b00), or stream ID associated with the data stream which the program will probably no longer access(S=0b10).

Programming Note

To maximize the utility of the Depth control mechanism, the architecture provides a hierarchy of three ways to program it. The DPFD field in the LPCR is used by the provisory/firmware to set a safe or appropriate default depth for unaware operating systems and applications. The DPFD field in the DSCR may be initialized by the aware OS and overwritten by an application via the OS-provided service when per stream control is unnecessary or unaffordable. The DEP field in the EA specification when TH=0b01010 may be used by the application to specify the depth on a per-stream basis.

The number of elements ahead of the point of stream use indicated by a given depth value may differ across implementations, as may the latency to bring a given element into the cache. To achieve optimum performance, some experimentation with different depth values may be necessary.

01011 The ***dcbt***/***dcbtst*** instruction provides a hint that describes certain attributes of a data stream.

The EA is interpreted as follows.

///	STRIDE	OFFSET	//	ID
0	32	50	56	60 63

Bit(s)	Definition
0:31	Reserved
32:49	**Stride** 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)×STRIDE×4 greater than or less than the effective address of the first element of the stream, depending on the direction specified for the stream.
50	Reserved
51:55	**Offset** 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 (EATRUNC || OFFSET || 0b00)).
56:59	Reserved
60:63	**Stream ID** (ID) Stream ID to use for this data stream.

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.

Engineering Note

Because the stride is specified in terms of words, the hardware should use a word-address register for calculating the effective address of the next element in a stream. When a stream is initiated, bits 0:56 of this register are loaded with EATRUNC (specified by a ***dcbt***/***dcbtst*** instruction with TH=0b01000). Bits 57:61 are loaded with either OFFSET (specified by a ***dcbt***/***dcbtst*** instruction with TH=0b01011), or zeros (if OFFSET is not specified for the stream). For each subsequent element, this word-address register is incremented or decremented either by STRIDE (specified by a ***dcbt***/***dcbtst*** instruction with TH=0b01011) or by 32 (if STRIDE is not specified for the stream).

Architecture Note

In the future, bits 56:59 of the EA when TH=01011 may be used to specify the element size of a stream. Knowledge of a stream's element size enables hardware to detect when an element in the stream spans adjacent cache blocks.

Architecture Note

The fact that bit 59 of the EA encodings is reserved permits the ID field to be enlarged to five bits in the future if that proves desirable.

If the specified stream ID value is greater than m−1, where m is the number of stream IDs provided by the implementation, and either (a) TH=0b01000 or TH=0b01011, or (b) TH=0b01010 with GO=0 and S≠0b11, no hint is provided by the instruction.

Engineering Note

It is suggested that undefined values of the TH field be treated as if they were 0b00000. This facilitates backward compatibility if these values are defined to provide new hints in the future.

Similarly, it is suggested that the reserved value of the S field of the EA encoding (TH=0b01010) be treated as if it were 0b00.

The following terminology is used to describe the state of a data stream. Except as described in the paragraph after the next paragraph, the state of a data stream at a given time is determined by the most recently provided hint(s) for the stream.

- A data stream for which only descriptive hints have been provided (by ***dcbt***/***dcbtst*** instructions with TH=0b01000 and UG=0, TH=0b01010 and GO=0 and S=0b00, and/or with TH=0b01011) is said to be "nascent". A nascent data stream for which all relevant descriptive hints have been provided (by the ***dcbt***/***dcbtst*** usages listed in the preceding sentence) is considered to be "completely described". The order of descriptive hints with respect to one another is unimportant.
- A data stream for which a hint has been provided (by a ***dcbt***/***dcbtst*** instruction with TH=0b01000 and UG=1 or ***dcbt*** with TH=0b01010 and GO=1) that the program will probably soon access it is said to be "active".
- A data stream that is either nascent or active is considered to "exist".
- A data stream for which a hint has been provided (e.g., by a ***dcbt*** instruction with TH=0b01010 and S≠0b00) that the program will probably no longer access it is considered no longer to exist.

The hint provided by a ***dcbt***/***dcbtst*** instruction with TH=0b01000 and UG=1 implicitly includes a hint that the program will probably no longer access the data stream (if any) previously associated with the specified stream ID. The hint provided by a ***dcbt***/***dcbtst*** instruction with TH=0b01000 and UG=0, or with TH=0b01010 and GO=0 and S=0b00, or with TH=0b01011 implicitly includes a hint that the program will probably no longer access the *active* data stream (if any) previously associated with the specified stream ID.

If a data stream is specified without using a ***dcbt***/***dcbtst*** instruction with TH=0b01010 and GO=0 and S=0b00, then the number of elements in the stream is unlimited, and the program's need for each element of the stream is not likely to be transient. If a data stream is specified without using a ***dcbt***/***dcbtst*** instruction with

TH=0b01011, then the stream will access consecutive units of storage.

A context switch on a particular thread causes all existing data streams for that thread to cease to exist. The mechanism of detection of a context switch is implementation-dependent. In addition, depending on the implementation, certain conditions and events may cause an existing data stream to cease to exist; for example, in some implementations an existing data stream ceases to exist when it comes to the end of a page.

Engineering Note

For the purposes of software data streams, any change to PIDR or LPIDR will usually be defined as a context switch which will clear all existing streams for a thread. Some implementations may also define changes to certain fields of MSR as a stream-clearing context switch.

Programming Note

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

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$\neq$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.

Programming Note

This Programming Note describes several aspects of using the data stream variants of the ***dcbt*** and ***dcbtst*** instructions.

- A non-transient data stream having unlimited length and which will access consecutive units in storage can be completely specified, including providing the hint that the program will probably soon access it, using one ***dcbt*** instruction. The corresponding specification for a data stream having other attributes requires two or three ***dcbt***/***dcbtst*** instructions to describe the stream and one additional ***dcbt*** instruction to start the stream. However, one ***dcbt*** instruction with TH=0b01010 and GO=1 can apply to a set of the data streams described in the preceding sentence, so the corresponding specification for n such data streams requires 2×n to 3×n ***dcbt***/***dcbtst*** instructions plus one ***dcbt*** instruction. (There is no need to execute a ***dcbt***/***dcbtst*** instruction with TH=0b01010 and S=0b10 for a given stream ID before using the stream ID for a new data stream; the implicit portion of the hint provided by ***dcbt***/***dcbtst*** instructions that describe data streams suffices.)
- If it is desired that the hint provided by a given ***dcbt***/***dcbtst*** instruction be provided in program order with respect to the hint provided by another ***dcbt***/***dcbtst*** instruction, the two instructions must be separated by an ***eieio*** instruction. For example, if a ***dcbt*** instruction with TH=0b01010 and GO=1 is intended to indicate that the program will probably soon access nascent data streams described (completely) by preceding ***dcbt***/***dcbtst*** instructions, and is intended *not* to indicate that the program will probably soon access nascent data streams described (completely) by following ***dcbt***/***dcbtst*** instructions, an ***eieio*** instruction must separate the ***dcbt*** instruction with GO=1 from the preceding ***dcbt***/***dcbtst*** instructions, and another ***eieio*** instruction must separate that ***dcbt*** instruction from the following ***dcbt***/***dcbtst*** instructions.
- In practice, the second ***eieio*** described above can sometimes be omitted. For example, if the program consists of an outer loop that contains the ***dcbt***/***dcbtst*** instructions and an inner loop that contains the *Load* or *Store* instructions that access the data streams, the characteristics of the inner loop and of the implementation's branch prediction mechanisms may make it highly unlikely that hints corresponding to a given iteration of the outer loop will be provided out of program order with respect to hints corresponding to the previous iteration of the outer loop. (Also, any providing of hints out of program order affects only performance, not program correctness.)
- To mitigate the effects of context switches on data streams, it may be desirable to specify a given "logical" data stream as a sequence of shorter, component data streams. Similar considerations apply to conditions and events that, depending on the implementation, may cause an existing data stream to cease to exist; for example, in some implementations an existing data stream ceases to exist when it comes to the end of a virtual page.
- If it is desired to specify data streams without regard to the number of stream IDs provided by the implementation, stream IDs should be assigned to data streams in order of decreasing stream importance (stream ID 0 to the most important stream, stream ID 1 to the next most important stream, etc.). This order ensures that the hints for the most important data streams will be provided.
- In practice, it is usually the case that it is preferable for data streams of the main or parent application running on a particular thread to be prioritized over data streams of a leaf or child application or function call. As an example, a library function call would usually fall under the category of a leaf function whose data streams should be treated with lower priority than those of the parent. As such, it is advised that programmers writing library functions or other functions which are expected to have leaf routing use the lowest stream priority possible. For example, if a leaf-type function instantiates two data streams, best practice is to use stream ID 14 for its most important data stream, and stream ID 15 for its second most important stream.

Engineering Note

In designing the data cache, account should be taken of the fact that programs that use the data stream variants of the ***dcbt***/***dcbtst*** instructions are likely to use multiple data streams at the same time. In levels of cache shared among processors, allowance should also be made for programs on the processors having active streams concurrently. While an individual program can allocate streams in such a way as to minimize collisions in congruence classes of the cache, separate programs cannot control how their streams interact. The cache allocation and replacement policy should attempt to minimize the displacement of elements of one program's transient streams by elements from the streams of another program running on a different processor.

Engineering Note

In a multithreaded system, a thread is not guaranteed to have 16 physical sets of data stream resources available for its own use. In addition, multiple functions on a specific thread may attempt to simultaneously use the same physical data stream resource. Implementations will usually manage data stream resource conflicts based on the following rules:

- One thread's specified data streams will never overwrite another thread's streams.
- On a single thread, if a newly specified data stream is mapped to a hardware resource which is already in use by another data stream, the new stream will only replace the old stream if the new stream is of equal or greater importance than the old stream (with stream ID 0 taken as highest importance and stream ID 15 taken as lowest importance).
- If the new stream is of lesser importance than the old stream, it will not be serviced by the processor.

TH=0b10000

If TH=0b10000, the ***dcbt*** instruction provides a hint that the program will probably soon load from the block containing the byte addressed by EA, and that the program's need for the block will be transient (i.e., the time interval during which the program accesses the block is likely to be short).

Programming Note

The processor's response to the hint that access to the block will be transient is to prefetch data into the cache hierarchy in a way that minimizes the displacement of data that has not been identified as transient.

Engineering Note

A possible implementation for this instruction is to mark the addressed block as LRU in the L2 cache when it is first loaded.

TH=0b10001

If TH=0b10001, the ***dcbt*** instruction provides a hint that the program will probably not access the block containing the byte addressed by EA for a relatively long period of time.

Engineering Note

A possible implementation for this hint is to mark the addressed block as LRU and ensure that it is not placed in a cache of a higher level when it is evicted from its current cache location.

Data Cache Block Touch X-form

dcbt RA,RB,TH

31	TH	RA	RB	278	/
0	6	11	16	21	31

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

The ***dcbt*** instruction provides a hint that describes a block or data stream to which the program may perform a *Load* access. The instruction is also used to indicate imminent access or end of access to described load and store data streams. A hint that the program will probably soon load from a given storage location is ignored if the location is Caching Inhibited or Guarded.

The only operation that is "caused" by the ***dcbt*** instruction is the providing of the hint. The actions (if any) taken by the processor in response to the hint are not considered to be "caused by" or "associated with" the ***dcbt*** instruction (e.g., ***dcbt*** is considered not to cause any data accesses). No means are provided by which software can synchronize these actions with the execution of the instruction stream. For example, these actions are not ordered by memory barriers.

The ***dcbt*** instruction may complete before the operation it causes has been performed.

The nature of the hint depends, in part, on the value of the TH field, as specified at the beginning of this section. If TH$\neq$0b01010 and TH$\neq$0b01011, this instruction is treated as a *Load* (see Section 4.3), except that the system data storage error handler is not invoked, and reference and change recording need not be done.

Special Registers Altered:

None

Extended Mnemonics:

Extended mnemonics are provided for the *Data Cache Block Touch* instruction so that it can be coded with the TH value as the last operand for all categories, and so that the transient hint can be specified without coding the TH field explicitly.

Extended mnemonic:		Equivalent to:	
dcbt	RA,RB	dcbt	RA,RB,0b00000
dcbtt	RA,RB	dcbt	RA,RB,0b10000
dcbna	RA,RB	dcbt	RA,RB,0b10001
dcbtds	RA,RB,TH	dcbt	RA,RB,TH

(TH = 0b00000 OR (TH $\geq$ 0b01000 & TH $\leq$ 0b01111))

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.

Data Cache Block Touch for Store X-form

dcbtst RA,RB,TH

31	TH	RA	RB	246	/
0	6	11	16	21	31

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

The ***dcbtst*** instruction provides a hint that describes a block or data stream to which the program may perform a *Store* access, or indicates the expected use thereof. A hint that the program will soon store to a given storage location is ignored if the location is Caching Inhibited or Guarded.

The only operation that is "caused by" the ***dcbtst*** instruction is the providing of the hint. The actions (if any) taken by the processor in response to the hint are not considered to be "caused by" or "associated with" the ***dcbtst*** instruction (e.g., ***dcbtst*** is considered not to cause any data accesses). No means are provided by which software can synchronize these actions with the execution of the instruction stream. For example, these actions are not ordered by memory barriers.

The ***dcbtst*** instruction may complete before the operation it causes has been performed.

The nature of the hint depends, in part, on the value of the TH field, as specified at the beginning of this section. If TH$\neq$0b01010 and TH$\neq$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 mnemonic:	Equivalent to:
`dcbtstds RA,RB,TH` (TH = 0b00000 OR (TH $\geq$ 0b01000 & TH $\leq$ 0b01111))	`dcbtst RA,RB,TH`
`dcbtstt RA,RB`	`dcbtst RA,RB,0b10000`

Programming Note

See the Programming Notes at the beginning of this section.

Engineering Note

Executing ***dcbtst*** does not cause the corresponding cache blocks to be considered to be modified in the data cache.

Engineering Note

In designing the data cache, account should be taken of the fact that programs that use ***dcbt*** or ***dcbtst*** are likely to contain multiple instances of the instruction preceding the *Load* or *Store* instructions that refer to the corresponding cache blocks.

Data Cache Block set to Zero X-form

dcbz RA,RB

31	///	RA	RB	1014	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + (RB)
n ← block size (bytes)
m ← log2(n)
ea ← EA0:63-m || m0
MEM(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

dcbz does not cause the block to exist in the data cache if the block is in storage that is Caching Inhibited.

For storage that is neither Write Through Required nor Caching Inhibited, ***dcbz*** provides an efficient means of setting blocks of storage to zero. It can be used to initialize large areas of such storage, in a manner that is likely to consume less memory bandwidth than an equivalent sequence of *Store* instructions.

For storage that is either Write Through Required or Caching Inhibited, ***dcbz*** is likely to take significantly longer to execute than an equivalent sequence of *Store* instructions. For example, on some implementations ***dcbz*** for such storage may cause the system alignment error handler to be invoked; on such implementations the system alignment error handler sets the specified block to zero using Store instructions.

See Section 6.9.1 of Book III for additional information about ***dcbz***.

Engineering Note

If the specified block is in storage that is neither Write Through Required nor Caching Inhibited and is not already in the data cache, establishing the block in the data cache without fetching it from main storage may provide the best performance.

If the specified block is in storage that is either Write Through Required or Caching Inhibited, an Alignment exception may be generated.

If ***dcbz*** causes the specified block to be established in the data cache without being fetched from main storage, the contents of any byte of the cache block must not be made available to another processor or mechanism until that byte has been set to zero.

Data Cache Block Store X-form

dcbst RA,RB

31	///	RA	RB	54	/
0	6	11	16	21	31

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

If the block containing the byte addressed by EA is in storage that is Memory Coherence Required and a block containing the byte addressed by EA is in the data cache of any processor and any locations in the block are considered to be modified there, those locations are written to main storage, additional locations in the block may be written to main storage, and the block ceases to be considered to be modified in that data cache.

If the block containing the byte addressed by EA is in storage that is not Memory Coherence Required and the block is in the data cache of this processor and any locations in the block are considered to be modified there, those locations are written to main storage, additional locations in the block may be written to main storage, and the block ceases to be considered to be modified in that data cache.

The function of this instruction is independent of whether the block containing the byte addressed by EA is in storage that is Write Through Required or Caching Inhibited.

This instruction is treated as a *Load* (see Section 4.3), except that reference and change recording need not be done.

Special Registers Altered:

None

Programming Note

Data Cache Block Store to Persistent Storage is encoded as a variant of *Data Cache Block Flush*. This was necessary so that the instruction has the correct behavior on processors that implement Version 3.0C. The extended mnemonic (***dcbstps***) indicates the intended function. (There is expected to be no interest in attaching persistent storage to processors that comply with versions of the architecture that precede Version 3.0C.)

Data Cache Block Flush X-form

dcbf RA,RB,L

31	//	L	RA	RB	86	/
0	6	8	11	16	21	31

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

L=0

If the block containing the byte addressed by EA is in storage that is Memory Coherence Required and a block containing the byte addressed by EA is in the data cache of any processor and any locations in the block are considered to be modified there, those locations are written to main storage and additional locations in the block may be written to main storage. The block is invalidated in the data caches of all processors.

If the block containing the byte addressed by EA is in storage that is not Memory Coherence Required and the block is in the data cache of this processor and any locations in the block are considered to be modified there, those locations are written to main storage and additional locations in the block may be written to main storage. The block is invalidated in the data cache of this processor.

L=1 (“dcbf local”)

The L=1 form of the ***dcbf*** instruction permits a program to limit the scope of the “flush” operation to the data cache of this processor. If the block containing the byte addressed by EA is in the data cache of this processor, it is removed from this cache. The coherence of the block is maintained to the extent required by the Memory Coherence Required storage attribute.

L = 3 (“dcbf local primary”)

The L=3 form of the ***dcbf*** instruction permits a program to limit the scope of the “flush” operation to the primary data cache of this processor. If the block containing the byte addressed by EA is in the primary data cache of this processor, it is removed from this cache. The coherence of the block is maintained to the extent required by the Memory Coherence Required storage attribute.

L = 4 (“data cache block flush to persistent storage”)

The L=4 form of the ***dcbf*** instruction performs all of the functions of ***dcbf*** with L=0. After all writes to main storage, caused by these functions, have updated main storage, if the block maps to main storage that is backed by persistent storage then all locations in the block in main storage that are considered to be modified relative to persistent storage are written to persistent storage and additional locations in the block in main storage may be written to persistent storage.

L = 6 (“data cache block store to persistent storage”)

The L=6 form of the ***dcbf*** instruction performs all of the functions of ***dcbst***. After all writes to main storage, caused by these functions, have updated main storage, if the block maps to main storage that is backed by persistent storage then all locations in the block in main storage that are considered to be modified relative to persistent storage are written to persistent storage and additional locations in the block in main storage may be written to persistent storage.

Programming Note

This form of the ***dcbf*** instruction is considered to be a functional extension of ***dcbst***, and its extended mnemonic reflects that association.

For the L operand, the values 2, 5, and 7 are reserved. The results of executing a ***dcbf*** instruction with L=2, 5, or 7 are boundedly undefined.

Engineering Note

dcbf with an L value of 2 should be treated as if the value were 0.

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 A “Assembler Extended Mnemonics ” on page 1042. The extended mnemonics are shown below.

Extended mnemonic:		Equivalent to:	
dcbf	RA,RB	dcbf	RA,RB,0
dcbfl	RA,RB	dcbf	RA,RB,1
dcbflp	RA,RB	dcbf	RA,RB,3
dcbfps	RA,RB	dcbf	RA,RB,4
dcbstps	RA,RB	dcbf	RA,RB,6

Except in the ***dcbf*** instruction description in this section, references to “***dcbf***” in Books I-III imply L=0 unless otherwise stated or obvious from context; “***dcbfl***” is used for L=1, “***dcbflp***” is used for L=3, “***dcbfps***” is used for L=4, and “***dcbstps***” is used for L=6.

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 or L=4.

Engineering Note

dcbf with L=1 requires caches that are private to this processor to be flushed. Caches shared with other processors need not be affected provided that the requirements of memory coherence are satisfied.

4.3.2.1 Obsolete Data Cache Instructions

The *Data Stream Touch* (***dst***), *Data Stream Touch for Store* (***dststst***), 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 MSR_{VEC} (see Book III).

Programming Note

These instructions merely provided hints, and thus were permitted to be treated as no-ops even on processors that implemented them.

The treatment of these instructions is independent of whether other Vector instructions are available because, on processors that implemented the instructions, the instructions were available even when other Vector instructions were not.

The extended mnemonics for these instructions were ***dstt***, ***dststt***, and ***dssall***.

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 ***or***:

Extended mnemonic:	Equivalent to:
`miso`	`or 26,26,26`

“miso” is short for “make it so.”

Programming Note

This form of the ***or*** instruction can be used to reduce latency in producer-consumer applications by requesting that modified data be made visible to other processors quickly. In this example it is assumed that the base register is GPR3.

Producer:

```
  addi r1,r0,0x1234
  sth r1,0x1000(r3) # store data value 0x1234
  lwsync            # order 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:

```
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
```

Engineering Note

Implementation of ***miso*** should attempt to order the hint relative to stores caused by preceding accesses. Implementations may elect to treat ***miso*** as a no-op.

Ways to implement ***miso*** include the following.

- In a store queue that has potentially different priorities for forwarding stores, the ***miso*** instruction could increase the priority of stores caused by preceding instructions.
- In a store queue that holds stores to allow for gathering, the ***miso*** instruction could release the hold and allow the stores to proceed to be performed with respect to other processors.

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 from one storage location to another. Depending on the platform configuration, other alternatives such as posting a work element to a device's queue in control memory or copying a block to another system may be supported. The facility uses pairs of instructions, ***copy*** followed by ***paste.***, to define the data transfers. (See Section 1.7.1.1, "Storage Ordering of Copy/Paste-Initiated Data Transfers" for the memory model characteristics of these data transfers.)

The requirements for use of the platform-specific functionality may vary across implementations. The specific details are beyond the scope of the Architecture. Authority to use a device is generally established through a call to the hypervisor. The format of the work element is device-specific. Mappings for transfers between systems are generally managed by the operating system. Each transfer preserves the order of bytes in storage and is not affected by the endian mode of the processor.

Since the buffer that holds the block until a data transfer is performed is hidden state (cannot be saved and restored) and there is no way to save the state of the ***copy***, any disruption of program execution (e.g. interrupts, event-based branch) has the potential to prevent the data transfer from completing correctly. The software that handles the disruption is responsible for executing ***cpabort*** to clear the state associated with an outstanding data transfer if it will use the Copy-Paste Facility itself or transfer control to another program that might use the facility prior to returning control to the original program.

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.

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

WARNING: In rare circumstances, ***paste.*** may falsely report successful completion when the copy-paste sequence is coded incorrectly. This may occur if the instruction sequence includes a redundant ***copy*** and the sequence is interrupted just prior to the redundant ***copy***. Since interrupts should be rare, any sequence that returns a false positive CR0 value should fail for most executions.

Programming Note

It is always best to avoid unnecessary instructions between the ***copy*** and the ***paste.***

Successful transfers are indicated when ***paste.*** returns 0b001x in CR0. Transient errors (a copy-paste sequence error, a translation conflict (***slbie***[***g***], ***slbia***, ***tlbie***[***l***]) during the transfer, or an implementation-specific transient problem) are indicated by a CR0 value of 0b000x, indicating the sequence should be retried. (A sequence error is considered transient because it could have been caused by an interruption between the ***copy*** and ***paste.***.) Fatal errors unique to the Copy-Paste Facility (attempting to copy from control memory or attempting to use control memory that has not been properly configured) cause the system data storage error handler to be invoked when the (associated) ***paste.*** instruction is executed. ***paste.*** instructions that cause or report transient errors, fatal errors unique to the Copy-Paste Facility, or successful transfer completion reset the state of the facility so that a subsequent copy-paste sequence can begin with a clean slate.

Programming Note

For ***paste.*** to address space mapped to another system, OS control over that mapping is deemed to be a sufficient check on the configuration of the channel to the other system, so that a unique data storage interrupt type is not required.

Programming Note

A failure of a data transfer may be the result of a shortage of the resources required to complete the operation. (Such failures should only take place for transfers to control memory.) 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 control memory. If any other storage access (implicit or explicit, instruction or data) addresses control memory, 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

Control memory 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 X-form

copy RA,RB

31	///	1	RA	RB	774	/
0	6	10	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b +(RB)
copy_buffer ← memory(EA,128)||MEM_metadata(EA,128)
```

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

The 128 bytes in storage, and associated metadata, addressed by EA is loaded into the copy buffer.

If the EA is not a multiple of 128, the system alignment error handler is invoked.

If the specified block is in storage that is Caching Inhibited, the system data storage error handler is invoked

[] This instruction is treated as a *Load* (see Section 4.3, “Cache Management Instructions ”), except that the data transfer ordering is described in Section 1.7.1.1, “Storage Ordering of Copy/Paste-Initiated Data Transfers”.

Special Registers Altered:

None

Paste X-form

paste. RA,RB,L

31	///	L	RA	RB	902	1
0	6	10	11	16	21	31

```
if there was a copy-paste sequence error or
   a translation conflict
  CR0←0b000||XER_SO
else
  if RA = 0 then b ← 0
  else           b ← (RA)
  EA ← b +(RB)
  if L=1 then
    copy_buffer.md←0  /* clear metadata in buffer */
  post(MEM(EA,128)||MEM_metadata(EA,128))←copy_buffer
  wait for completion status
  if there was a data transfer problem
    CR0←0b000||XER_SO
  else
    CR0←0b001||XER_SO
clear the state of the Copy-Paste Facility
```

If there was a copy-paste sequence error or a translation conflict, set CR0 to indicate failure. Otherwise, continue as follows.

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

If L=1, then set metadata bits in the copy buffer to zero.

The physical target of the operation, and by implication the function to be performed, is determined by the real address that is translated from EA. If the real address is in the platform's system memory, a simple copy is performed. If the real address has an associated mapping to another system, the copy buffer is transmitted to the other system. If the real address is control memory for a device, the contents of the copy buffer is queued to the device. There is a wait for completion status on the data transfer. CR0 is set as follows based on the completion status.

CR0	*Description*
0b000||XER_{SO}	Data transfer failed due to a sequence error, a conflict with ***slbie***[***g***], ***slbia***, or ***tlbie***[***l***], or some implementation-specific problem.
0b001||XER_{SO}	Data transfer successful.

The state of the Copy-Paste Facility is cleared.

If MSR_{PR}=1, ***paste.*** with L=0 is an invalid form.

If the EA is not a multiple of 128, the system alignment error handler is invoked.

If the specified block is in storage that is Caching Inhibited, the system data storage error handler is invoked.

If the associated ***copy*** specified control memory or the ***paste.*** specifies control memory that was not properly configured, [] the data storage error handler will be invoked.

[] This instruction is treated as a Store (see Section 4.3, "Cache Management Instructions "), except that the data transfer ordering is described in Section 1.7.1.1, "Storage Ordering of Copy/Paste-Initiated Data Transfers".

Special Registers Altered:

Register	*Field(s)*
CR	CR0

Extended Mnemonics:

Extended Mnemonics for *Paste*:

Extended mnemonic:	Equivalent to:
`paste.  RA,RB`	`paste.  RA,RB,1`

Copy-Paste Abort X-form

cpabort

31	///	///	///	838	/
0	6	11	16	21	31

```
clear the state of the Copy-Paste Facility
```

The ***cpabort*** instruction causes a data transfer to fail if one is in progress.

Any pending errors in the Copy-Paste Facility are cleared and the state is reset to prepare for a new ***copy***.

Architecture Note

This instruction is added so that it is not necessary for hardware to be aware of a context switch, e.g. by observing a change to LPID || PID, and as a replacement for ***copy*** initializing the state of the facility.

Engineering Note

This instruction is intended to enable a general clean-up of the Copy/Paste Facility. It will be issued by the operating system or hypervisor during a task switch, much like a dummy *Store Conditional* instruction.

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 transfering the data from one processor's cache to another. The latency of such transfers can become the limiting factor in the performance of some environments. Rather than moving the data to the work, AMOs move the work to the data. The mental model is of an agent consisting of an execution unit and a work queue near memory that receives atomic update requests from all the processors in the system.

Despite that AMOs are performed at memory, their function is only defined for storage that is not Caching Inhibited. This is done so that software can transparently access the same data using normal loads and stores. But furthermore, AMOs generally behave as typical explicit storage accesses performed by the thread, with respect to storage access ordering. The few complications are described below. Since the performance advantage of AMOs derives from avoiding time of flight through cache hierarchies, software should avoid frequent mixing of normal loads and stores 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 *Load* instructions and fixed-point *Store* instructions, and any reference elsewhere in the architecture to fixed-point *Load* or fixed-point *Store* instructions apply to these instructions as well, unless otherwise stated or obvious from context. For example, in order to perform an AMO, it is necessary to have both read and write access to the storage location. Another example is that the DAWR will detect a match if either Data Read or Data Write is selected. Yet another example is that a Trace interrupt will indicate both a *Load* and a *Store* instruction have been executed. Barrier action will be based on whether the barrier would give a load or a store the stronger ordering. The difference between the *Load Atomic* instructions and the *Store Atomic* instructions is simply that the *Load Atomic* instructions return a result to a GPR, while the *Store Atomic* instructions do not. In the RTL in the following subsections, the "lat" and "stat" functions represent the manipulations performed by the memory agent. The parameters shown are the maximum storage footprint, the maximum list of registers, and the function code that are provided to the agent. If the specified registers wrap (e.g. RT=R31 and RT+1=R0), the wrapping is permitted. Such an instruction is not an invalid form. Destructive encodings are also permitted (i.e. a LAT specified with RT=RA).

Except in this section, references to "atomic update" in Books I-III imply use of the *Load And Reserve* and *Store Conditional* instructions unless otherwise stated or obvious from context.

Programming Note

The best performance for the Atomic Memory Operations will be realized when the targeted storage locations are accessed only using AMOs. If it is necessary to perform other I=0 loads and stores to those addresses, the result will still be correct, but performance will suffer. In such circumstances, it is not helpful to performance to flush the data to memory using ***dcbf***.

Programming Note

Note that the descriptions of AMO operations are Endian independent. The only effect of Endian on these operations is the obvious one that byte significance within an individual datum reflects the Endian mode.

Engineering Note

There is significant variation in the functions provided by Load Atomic and by Store Atomic. For example, the number of GPR operands varies. Hardware may choose to send a superset of GPRs to the memory agent rather than send the number of operands required by the specified function.

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 4.2.

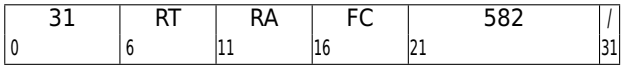
Function Code	GPR operands	Storage operands	Function name	RTL
00000	RT, RT+1	mem(EA,s)	Fetch and Add	t ← mem(EA, s) t2 ← t + (RT+1) mem(EA,s) ← t2 RT ← t
00001	RT, RT+1	mem(EA,s)	Fetch and XOR	t ← mem(EA, s) t2 ← t ⊕ (RT+1) mem(EA,s) ← t2 RT ← t
00010	RT, RT+1	mem(EA,s)	Fetch and OR	t ← mem(EA, s) t2 ← t | (RT+1) mem(EA,s) ← t2 RT ← t
00011	RT, RT+1	mem(EA,s)	Fetch and AND	t ← mem(EA, s) t2 ← t & (RT+1) mem(EA,s) ← t2 RT ← t
00100	RT, RT+1	mem(EA,s)	Fetch and Maximum Unsigned	t ← mem(EA, s) if (RT+1) $>^{u}$ t then mem(EA,s) ← (RT+1) RT ← t
00101	RT, RT+1	mem(EA,s)	Fetch and Maximum Signed	t ← mem(EA, s) if (RT+1) > t then mem(EA,s) ← (RT+1) RT ← t
00110	RT, RT+1	mem(EA,s)	Fetch and Minimum Unsigned	t ← mem(EA, s) if (RT+1) $<^{u}$ t then mem(EA,s) ← (RT+1) RT ← t
00111	RT, RT+1	mem(EA,s)	Fetch and Minimum Signed	t ← mem(EA, s) if (RT+1) < t then mem(EA,s) ← (RT+1) RT ← t
01000	RT, RT+1	mem(EA,s)	Swap	t ← mem(EA, s) mem(EA,s) ← (RT+1) RT ← t
10000	RT, RT+1, RT+2	mem(EA,s)	Compare and Swap Not Equal	t ← mem(EA, s) if t != (RT+1) then mem(EA,s) ← (RT+2) RT ← t
11000	RT	mem(EA,s) mem(EA+s, s)	Fetch and Increment Bounded	t ← mem(EA, s) t2 ← mem(EA+s, s) if t != t2 then mem(EA,s) ← t+1; RT ← t else RT ← 1 << (s*8-1)
11001	RT	mem(EA,s) mem(EA+s, s)	Fetch and Increment Equal	t ← mem(EA, s) t2 ← mem(EA+s, s) if t = t2 then mem(EA,s) ← t+1; RT ← t else RT ← 1 << (s*8-1)
11100	RT	mem(EA-s,s) mem(EA, s)	Fetch and Decrement Bounded	t ← mem(EA, s) t2 ← mem(EA-s, s) if t != t2 then mem(EA,s) ← t-1; RT ← t else RT ← 1 << (s*8-1)

Notes:
s = operand size in number of bytes
Function codes not listed in this table are reserved.
For word atomics, only the least significant word of each source register is used, and the least significant word of the target register is updated with the result, while the upper word is set to zero.

Figure 4.2: Load Atomic function codes

Load Word Atomic X-form

lwat RT,RA,FC




```
if RA=0 then EA ← 0
else         EA ← (RA)
(RT32:63,mem(EA,4))← lat(mem(EA-4,12), RT+132:63,
RT+232:63, FC)
RT0:31←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 4.2 contains the defined function codes. An attempt to execute ***lwat*** specifying a reserved function code will cause the system data storage error handler to be invoked.

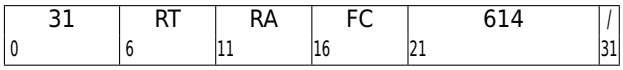
EA must be a multiple of 4, and the portion of mem(EA-4,12) accessed by the instruction must be contained within an aligned 32-byte block of storage. If either of these requirements is not satisfied, the system alignment error handler is invoked.

Special Registers Altered:

None

Load Doubleword Atomic X-form

ldat RT,RA,FC




```
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 4.2 contains the defined function codes. An attempt to execute ***ldat*** specifying a reserved function code will cause the system data storage error handler to be invoked.

EA must be a multiple of 8, and the portion of mem(EA-8,24) accessed by the instruction must be contained within an aligned 32-byte block of storage. If either of these requirements is not satisfied, the system alignment error handler is invoked.

Special Registers Altered:

None

4.5.2 Store Atomic

The Atomic Stores perform an atomic update to an aligned memory location. The manipulation performed on the memory value is determined by the function code (FC) specified by the instruction. The name of each function and its associated RTL are shown in Figure 4.3.

Function Code	GPR operands	Storage operands	Function name	RTL
00000	RS	mem(EA,s)	Store Add	t ← mem(EA, s) t2 ← t + (RS) mem(EA,s) ← t2
00001	RS	mem(EA,s)	Store XOR	t ← mem(EA, s) t2 ← t ⊕ (RS) mem(EA,s) ← t2
00010	RS	mem(EA,s)	Store OR	t ← mem(EA, s) t2 ← t | (RS) mem(EA,s) ← t2
00011	RS	mem(EA,s)	Store AND	t ← mem(EA, s) t2 ← t & (RS) mem(EA,s) ← t2
00100	RS	mem(EA,s)	Store Maximum Unsigned	t ← mem(EA, s) if (RS) $>^{u}$ t then mem(EA,s) ← (RS)
00101	RS	mem(EA,s)	Store Maximum Signed	t ← mem(EA, s) if (RS) > t then mem(EA,s) ← (RS)
00110	RS	mem(EA,s)	Store Minimum Unsigned	t ← mem(EA, s) if (RS) $<^{u}$ t then mem(EA,s) ← (RS)
00111	RS	mem(EA,s)	Store Minimum Signed	t ← mem(EA, s) if (RS) < t then mem(EA,s) ← (RS)

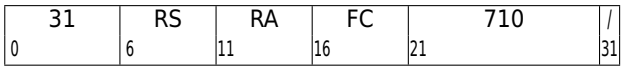
11000	RS	mem(EA,s)
		mem(EA+s, s)
			Store Twin	t ← mem(EA, s)
				t2 ← mem(EA+s, s)
				if t = t2 then
				  mem(EA,s) ← (RS)
				  mem(EA+s,s) ← (RS)

Notes:
s = operand size in number of bytes
Function codes not listed in this table are reserved.
For word atomics, only the least significant word of each source register is used.

Figure 4.3: Store Atomic function codes

Store Word Atomic X-form

stwat RS,RA,FC




```
if RA=0 then EA ← 0
else         EA ← (RA)
mem(EA,8)← stat(mem(EA,8), RS32:63, FC)
```

Let the effective address (EA) be (RA). Four or eight bytes of storage at EA are updated as specified by store atomic function code FC. Input operands are function code specific, and may include $RS_{32:63}$ and mem(EA,8).

Figure 4.3 contains the defined function codes. An attempt to execute ***stwat*** specifying a reserved function code will cause the system data storage error handler to be invoked.

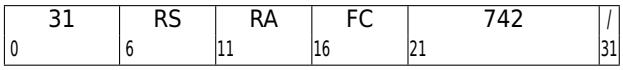
EA must be a multiple of 4, and the portion of mem(EA,8) accessed by the instruction must be contained within an aligned 32-byte block of storage. If either of these requirements is not satisfied, the system alignment error handler is invoked.

Special Registers Altered:

None

Store Doubleword Atomic X-form

stdat RS,RA,FC




```
if RA=0 then EA ← 0
else         EA ← (RA)
mem(EA,16)← stat(mem(EA,16), RS, FC)
```

Let the effective address (EA) be (RA). Eight or sixteen bytes of storage at EA are updated as specified by store atomic function code FC. Input operands are function code specific, and may include RS and mem(EA,16).

Figure 4.3 contains the defined function codes. An attempt to execute ***stdat*** specifying a reserved function code will cause the system data storage error handler to be invoked.

EA must be a multiple of 8, and the portion of mem(EA,16) accessed by the instruction must be contained within an aligned 32-byte block of storage. If either of these requirements is not satisfied, the system alignment error handler is invoked.

Special Registers Altered:

None

4.6 Synchronization Instructions

The synchronization instructions are used to ensure that certain instructions have completed before other instructions are initiated, or to control storage access ordering, or to support debug operations.

4.6.1 Instruction Synchronize Instruction

Instruction Synchronize XL-form

isync

19	///	///	///	150	/
0	6	11	16	21	31

Executing an ***isync*** instruction ensures that all instructions preceding the ***isync*** instruction have completed before the ***isync*** instruction completes, and that no subsequent instructions are initiated until after the ***isync*** instruction completes. It also ensures that all instruction cache block invalidations caused by ***icbi*** instructions preceding the ***isync*** instruction have been performed with respect to the processor executing the ***isync*** instruction, and then causes any prefetched instructions to be discarded.

Except as described in the preceding sentence, the ***isync*** instruction may complete before storage accesses associated with instructions preceding the ***isync*** instruction have been performed.

This instruction is context synchronizing (see Book III).

Special Registers Altered:

None

4.6.2 Load And Reserve and Store Conditional Instructions

The *Load And Reserve* and *Store Conditional* instructions can be used to construct a sequence of instructions that appears to perform an atomic update operation on an aligned storage location. See Section 1.7.2, “Atomic Update” for additional information about these instructions.

The *Load And Reserve* and *Store Conditional* instructions are fixed-point *Storage Access* instructions; see Section 3.3.1, “Fixed-Point Storage Access Instructions”, in Book I.

The storage location specified by the *Load And Reserve* and *Store Conditional* instructions must be in storage that is Memory Coherence Required if the location may be modified by another processor or mechanism. If the specified location is in storage that is Write Through Required or Caching Inhibited, the system data storage error handler is invoked.

The *Load And Reserve* instructions include an Exclusive Access hint (EH), which can be used to indicate that the instruction sequence being executed is implementing one of two types of algorithms:

Atomic Update (EH=0)
This hint indicates that the program is using a fetch and operate (e.g., fetch and add) or some similar algorithm and that all programs accessing the shared variable are likely to use a similar operation to access the shared variable for some time.

Exclusive Access (EH=1)
This hint indicates that the program is attempting to acquire a lock and if it succeeds, will perform another store to the lock variable (releasing the lock) before another program attempts to modify the lock variable.

Programming Note

The Memory Coherence Required attribute on other processors and mechanisms ensures that their stores to the reservation granule will cause the reservation created by the *Load And Reserve* instruction to be lost.

Programming Note

Because the *Load And Reserve* and *Store Conditional* instructions have implementation dependencies (e.g., the granularity at which reservations are managed), they must be used with care. The operating system should provide system library programs that use these instructions to implement the high-level synchronization functions (Test and Set, Compare and Swap, locking, etc.; see Appendix B) that are needed by application programs. Application programs should use these library programs, rather than use the *Load And Reserve* and *Store Conditional* instructions directly.

Programming Note

EH = 1 should be used when the program is obtaining a lock variable which it will subsequently release before another program attempts to perform a store to it. When contention for a lock is significant, using this hint may reduce the number of times a cache block is transferred between processor caches.

EH = 0 should be used when all accesses to a mutex variable are performed using an instruction sequence with *Load And Reserve* followed by *Store Conditional* (e.g., emulating atomic update primitives such as “Fetch and Add;” see Appendix B). The processor may use this hint to optimize the cache to cache transfer of the block containing the mutex variable, thus reducing the latency of performing an operation such as ‘Fetch and Add’.

[]

Engineering Note

Causing an Alignment exception if attempt is made to execute a *Load And Reserve* or *Store Conditional* instruction having an incorrectly aligned effective address facilitates the debugging of software.

Load Byte And Reserve Indexed X-form

lbarx RT,RA,RB,EH

31	RT	RA	RB	52	EH
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b +(RB)
RESERVE ← 1
RESERVE_LENGTH ← 1
RESERVE_ADDR ← real_addr(EA)
RT ← 56^0 || MEM(EA, 1)
```

Let the effective address (EA) be the sum (RA|0)+(RB). The byte in storage addressed by EA is loaded into $RT_{56:63}$. $RT_{0:55}$ are set to 0.

This instruction creates a reservation for use by a ***stbcx.*** or ***waitrsv*** instruction. A real address computed from the EA as described in Section 1.7.2.1 is associated with the reservation, and replaces any address previously associated with the reservation. A length of 1 byte is associated with the reservation, and replaces any length previously associated with the reservation.

The value of EH provides a hint as to whether the program will perform a subsequent store to the byte in storage addressed by EA before some other processor attempts to modify it.

- 0 Other programs might attempt to modify the byte in storage addressed by EA regardless of the result of the corresponding ***stbcx.*** instruction.
- 1 Other programs will not attempt to modify the byte in storage addressed by EA until the program that has acquired the lock performs a subsequent store releasing the lock.

Special Registers Altered:

None

Extended Mnemonics:

Extended Mnemonics for *Load Byte And Reserve Indexed*:

Extended mnemonic:	Equivalent to:
`lbarx RT,RA,RB`	`lbarx RT,RA,RB,0`

Programming Note

lbarx serves as both a basic and an extended mnemonic. The Assembler will recognize a ***lbarx*** mnemonic with four operands as the basic form, and a ***lbarx*** mnemonic with three operands as the extended form. In the extended form the EH operand is omitted and assumed to be 0.

Load Halfword And Reserve Indexed X-form

lharx RT,RA,RB,EH

31	RT	RA	RB	116	EH
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b +(RB)
RESERVE ← 1
RESERVE_LENGTH ← 2
RESERVE_ADDR ← real_addr(EA)
RT ← 48^0 || MEM(EA, 2)
```

Let the effective address (EA) be the sum (RA|0)+(RB). The halfword in storage addressed by EA is loaded into $RT_{48:63}$. $RT_{0:47}$ are set to 0.

This instruction creates a reservation for use by a ***sthcx.*** or ***waitrsv*** instruction. A real address computed from the EA as described in Section 1.7.2.1 is associated with the reservation, and replaces any address previously associated with the reservation. A length of 2 bytes is associated with the reservation, and replaces any length previously associated with the reservation.

The value of EH provides a hint as to whether the program will perform a subsequent store to the halfword in storage addressed by EA before some other processor attempts to modify it.

- 0 Other programs might attempt to modify the halfword in storage addressed by EA regardless of the result of the corresponding ***sthcx.*** instruction.
- 1 Other programs will not attempt to modify the halfword in storage addressed by EA until the program that has acquired the lock performs a subsequent store releasing the lock.

EA must be a multiple of 2. If it is not, either the system alignment error handler is invoked or the results are boundedly undefined.

Special Registers Altered:

None

Extended Mnemonics:

Extended Mnemonics for *Load Halfword And Reserve Indexed*:

Extended mnemonic:	Equivalent to:
`lharx RT,RA,RB`	`lharx RT,RA,RB,0`

Programming Note

lharx serves as both a basic and an extended mnemonic. The Assembler will recognize a ***lharx*** mnemonic with four operands as the basic form, and a ***lharx*** mnemonic with three operands as the extended form. In the extended form the EH operand is omitted and assumed to be 0.

Load Word And Reserve Indexed X-form

lwarx RT,RA,RB,EH

31	RT	RA	RB	20	EH
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b +(RB)
RESERVE ← 1
RESERVE_LENGTH ← 4
RESERVE_ADDR ← real_addr(EA)
RT ← 32^0 || MEM(EA, 4)
```

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

This instruction creates a reservation for use by a ***stwcx.*** or ***waitrsv*** instruction. A real address computed from the EA as described in Section 1.7.2.1 is associated with the reservation, and replaces any address previously associated with the reservation. A length of 4 bytes is associated with the reservation, and replaces any length previously associated with the reservation.

The value of EH provides a hint as to whether the program will perform a subsequent store to the word in storage addressed by EA before some other processor attempts to modify it.

0 Other programs might attempt to modify the word in storage addressed by EA regardless of the result of the corresponding ***stwcx.*** instruction.

1 Other programs will not attempt to modify the word in storage addressed by EA until the program that has acquired the lock performs a subsequent store releasing the lock.

EA must be a multiple of 4. If it is not, either the system alignment error handler is invoked or the results are boundedly undefined.

Special Registers Altered:

None

Extended Mnemonics:

Extended Mnemonics for *Load Word And Reserve Indexed*:

Extended mnemonic:	Equivalent to:
`lwarx RT,RA,RB`	`lwarx RT,RA,RB,0`

Programming Note

lwarx serves as both a basic and an extended mnemonic. The Assembler will recognize a ***lwarx*** mnemonic with four operands as the basic form, and a ***lwarx*** mnemonic with three operands as the extended form. In the extended form the EH operand is omitted and assumed to be 0.

Store Byte Conditional Indexed X-form

stbcx. RS,RA,RB

31	RS	RA	RB	694	1
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + (RB)
if RESERVE then
  if RESERVE_LENGTH = 1 &
     RESERVE_ADDR = real_addr(EA) then
      MEM(EA, 1) ← (RS)56:63
      undefined_case  ← 0
      store_performed ← 1
   else
      z ← smallest real page size supported by implementation
      if RESERVE_ADDR ÷ z = real_addr(EA) ÷ z then
        undefined_case  ← 1
      else
        undefined_case  ← 0
        store_performed ← 0
else
  undefined_case  ← 0
  store_performed ← 0
if undefined_case then
  u1 ← undefined 1-bit value
  if u1 then
    MEM(EA, 1) ← (RS)56:63
  u2 ← undefined 1-bit value
  CR0 ← 0b00 || u2 || XERSO
else
  CR0 ← 0b00 || store_performed || XERSO
RESERVE ← 0
```

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

If a reservation exists, the length associated with the reservation is 1 byte, and the real storage location specified by the ***stbcx.*** is the same as the real storage location specified by the ***lbarx*** instruction that established the reservation, $(RS)_{56:63}$ are stored into the byte in storage addressed by EA.

If a reservation exists, and either the length associated with the reservation is not 1 byte or the real storage location specified by the ***stbcx.*** is not the same as the real storage location specified by the ***lbarx*** instruction that established the reservation, the following applies. Let z denote the smallest real page size supported by the implementation. If the real storage location specified by the ***stbcx.*** is in the same aligned z-byte block of real storage as the real storage location specified by the ***lbarx*** instruction that established the reservation, it is undefined whether $(RS)_{56:63}$ are stored into the byte in storage addressed by EA. Otherwise, no store is performed.

If a reservation does not exist, no store is performed.

CR Field 0 is set as follows. n is a 1-bit value that indicates whether the store was performed, except that if, per the preceding description, it is undefined whether the store is performed, the value of n is undefined (and need not reflect whether the store was performed).

$CR0_{LT\ GT\ EQ\ SO}$ = 0b00 || n || XER_{SO}

The reservation is cleared.

Special Registers Altered:

Register	*Field(s)*
CR	CR0

Store Halfword Conditional Indexed X-form

sthcx. RS,RA,RB

31	RS	RA	RB	726	1
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + (RB)
if RESERVE then
  if RESERVE_LENGTH = 2 &
     RESERVE_ADDR = real_addr(EA) then
      MEM(EA, 2) ← (RS)48:63
      undefined_case  ← 0
      store_performed ← 1
   else
      z ← smallest real page size supported by implementation
      if RESERVE_ADDR ÷ z = real_addr(EA) ÷ 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, 2) ← (RS)48:63
  u2 ← undefined 1-bit value
  CR0 ← 0b00 || u2 || XERSO
else
  CR0 ← 0b00 || store_performed || XERSO
RESERVE ← 0
```

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

If a reservation exists, the length associated with the reservation is 2 bytes, and the real storage location specified by the ***sthcx.*** is the same as the real storage location specified by the ***lharx*** instruction that established the reservation, $(RS)_{48:63}$ are stored into the halfword in storage addressed by EA.

If a reservation exists, and either the length associated with the reservation is not 2 bytes or the real storage location specified by the ***sthcx.*** is not the same as the real storage location specified by the ***lharx*** instruction that established the reservation, the following applies. Let z denote the smallest real page size supported by the implementation. If the real storage location specified by the ***sthcx.*** is in the same aligned z-byte block of real storage as the real storage location specified by the ***lharx*** instruction that established the reservation, it is undefined whether $(RS)_{48:63}$ are stored into the halfword in storage addressed by EA. Otherwise, no store is performed.

If a reservation does not exist, no store is performed.

CR Field 0 is set as follows. n is a 1-bit value that indicates whether the store was performed, except that if, per the preceding description, it is undefined whether the store is performed, the value of n is undefined (and need not reflect whether the store was performed).

$CR0_{LT\ GT\ EQ\ SO} = 0b00 \mid\mid n \mid\mid XER_{SO}$

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:

Register	*Field(s)*
CR	CR0

Store Word Conditional Indexed X-form

stwcx. RS,RA,RB

31	RS	RA	RB	150	1
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + (RB)
if RESERVE then
  if RESERVE_LENGTH = 4 &
     RESERVE_ADDR = real_addr(EA) then
      MEM(EA, 4) ← (RS)32:63
      undefined_case  ← 0
      store_performed ← 1
   else
      z ← smallest real page size supported by implementation
      if RESERVE_ADDR ÷ z = real_addr(EA) ÷ z then
        undefined_case  ← 1
      else
        undefined_case  ← 0
        store_performed ← 0
else
  undefined_case  ← 0
  store_performed ← 0
if undefined_case then
  u1 ← undefined 1-bit value
  if u1 then
    MEM(EA, 4) ← (RS)32:63
  u2 ← undefined 1-bit value
  CR0 ← 0b00 || u2 || XERSO
else
  CR0 ← 0b00 || store_performed || XERSO
RESERVE ← 0
```

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

If a reservation exists, the length associated with the reservation is 4 bytes, and the real storage location specified by the ***stwcx.*** is the same as the real storage location specified by the ***lwarx*** instruction that established the reservation, $(RS)_{32:63}$ are stored into the word in storage addressed by EA.

If a reservation exists, and either the length associated with the reservation is not 4 bytes or the real storage location specified by the ***stwcx.*** is not the same as the real storage location specified by the ***lwarx*** instruction that established the reservation, the following applies. Let z denote the smallest real page size supported by the implementation. If the real storage location specified by the ***stwcx.*** is in the same aligned z-byte block of real storage as the real storage location specified by the ***lwarx*** instruction that established the reservation, it is undefined whether $(RS)_{32:63}$ are stored into the word in storage addressed by EA. Otherwise, no store is performed.

If a reservation does not exist, no store is performed.

CR Field 0 is set as follows. n is a 1-bit value that indicates whether the store was performed, except that if, per the preceding description, it is undefined whether the store is performed, the value of n is undefined (and need not reflect whether the store was performed).

$CR0_{LT\ GT\ EQ\ SO} = 0b00 \ || \ n \ || \ XER_{SO}$

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:

Register	*Field(s)*
CR	CR0

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

31	RT	RA	RB	84	EH
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b +(RB)
RESERVE ← 1
RESERVE_LENGTH ← 8
RESERVE_ADDR ← real_addr(EA)
RT ← MEM(EA, 8)
```

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

This instruction creates a reservation for use by a ***stdcx.*** or ***waitrsv*** instruction. A real address computed from the EA as described in Section 1.7.2.1 is associated with the reservation, and replaces any address previously associated with the reservation. A length of 8 bytes is associated with the reservation, and replaces any length previously associated with the reservation.

The value of EH provides a hint as to whether the program will perform a subsequent store to the doubleword in storage addressed by EA before some other processor attempts to modify it.

- 0 Other programs might attempt to modify the doubleword in storage addressed by EA regardless of the result of the corresponding ***stdcx.*** instruction.
- 1 Other programs will not attempt to modify the doubleword in storage addressed by EA until the program that has acquired the lock performs a subsequent store releasing the lock.

EA must be a multiple of 8. If it is not, either the system alignment error handler is invoked or the results are boundedly undefined.

Special Registers Altered:

None

Extended Mnemonics:

Extended Mnemonics for *Load Doubleword And Reserve Indexed*:

Extended mnemonic:	Equivalent to:
ldarx RT,RA,RB	ldarx RT,RA,RB,0

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

31	RS	RA	RB	214	1
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + (RB)
if RESERVE then
  if RESERVE_LENGTH = 8 &
     RESERVE_ADDR = real_addr(EA) then
      MEM(EA, 8) ← (RS)
      undefined_case  ← 0
      store_performed ← 1
   else
      z ← smallest real page size supported by implementation
      if RESERVE_ADDR ÷ z = real_addr(EA) ÷ z then
        undefined_case  ← 1
      else
        undefined_case  ← 0
        store_performed ← 0
else
  undefined_case  ← 0
  store_performed ← 0
if undefined_case then
  u1 ← undefined 1-bit value
  if u1 then
    MEM(EA, 8) ← (RS)
  u2 ← undefined 1-bit value
  CR0 ← 0b00 || 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 8 bytes, and the real storage location specified by the ***stdcx.*** is the same as the real storage location specified by the ***ldarx*** instruction that established the reservation, (RS) is stored into the doubleword in storage addressed by EA.

If a reservation exists, and either the length associated with the reservation is not 8 bytes or the real storage location specified by the ***stdcx.*** is not the same as the real storage location specified by the ***ldarx*** instruction that established the reservation, 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).

$CR0_{LT\ GT\ EQ\ SO}$ = 0b00 || n || XER_{SO}

The reservation is cleared.

EA must be a multiple of 8. If it is not, either the system alignment error handler is invoked or the results are boundedly undefined.

Special Registers Altered:

Register	*Field(s)*
CR	CR0

4.6.2.2 128-bit Load And Reserve and Store Conditional Instructions

For ***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 *Load Quadword and Reserve* instruction RA $\neq$ RTp+1 and RB $\neq$ RTp+1.

For ***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.

Load Quadword And Reserve Indexed X-form

lqarx RTp,RA,RB,EH

31	RTp	RA	RB	276	EH
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b +(RB)
RESERVE ← 1
RESERVE_LENGTH ← 16
RESERVE_ADDR ← real_addr(EA)
RTp ← 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 ***stqcx.*** or ***waitrsv*** instruction. A real address computed from the EA as described in Section 1.7.2.1 is associated with the reservation, and replaces any address previously associated with the reservation. A length of 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 ***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.

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

Extended Mnemonics:

Extended Mnemonics for *Load Quadword And Reserve Indexed*:

Extended mnemonic:	Equivalent to:
`lqarx RTp,RA,RB`	`lqarx RTp,RA,RB,0`

Programming Note

lqarx serves as both a basic and an extended mnemonic. The Assembler will recognize a ***lqarx*** mnemonic with four operands as the basic form, and a ***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

stqcx. RSp,RA,RB

31	RSp	RA	RB	182	1
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + (RB)
if RESERVE then
  if RESERVE_LENGTH = 16 &
     RESERVE_ADDR = real_addr(EA) then
      MEM(EA, 16) ← (RSp)
      undefined_case  ← 0
      store_performed ← 1
   else
      z ← smallest real page size supported by implementation
      if RESERVE_ADDR ÷ z = real_addr(EA) ÷ z then
        undefined_case  ← 1
      else
        undefined_case  ← 0
        store_performed ← 0
else
  undefined_case  ← 0
  store_performed ← 0
if undefined_case then
  u1 ← undefined 1-bit value
  if u1 then
    MEM(EA, 16) ← (RSp)
  u2 ← undefined 1-bit value
  CR0 ← 0b00 || u2 || XER[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 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, (RSp) is stored into the quadword in storage addressed by EA.

If a reservation exists, and either the length associated with the reservation is not 16 bytes or the real storage location specified by the ***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 (RSp) is stored into the quadword in storage addressed by EA. Otherwise, no store is performed.

If a reservation does not exist, no store is performed.

CR Field 0 is set as follows. n is a 1-bit value that indicates whether the store was performed, except that if, per the preceding description, it is undefined whether the store is performed, the value of n is undefined (and need not reflect whether the store was performed).

$CR0_{LT\ GT\ EQ\ SO}$ = 0b00 || n || XER_{SO}

The reservation is cleared.

EA must be a multiple of 16. If it is not, either the system alignment error handler is invoked or the results are boundedly undefined.

If RSp is odd, the instruction form is invalid.

Special Registers Altered:

Register	*Field(s)*
CR	CR0

4.6.3 Memory Barrier Instructions

The *Memory Barrier* instructions can be used to control the order in which storage accesses and data transfers are performed. Additional information about these instructions and about related aspects of storage management can be found in Book III.

Synchronize X-form

sync L,SC

31	//	L	///	SC	///	598	/
0	6	8	11	14	16	21	31

```
if SC≠0 then switch(SC)
     case(1): stncisync
     case(2): stcisync
     case(3): stsync
else switch(L)
     case(0): hwsync
     case(1): lwsync
     case(2): ptesync
     case(4): phwsync
     case(5): plwsync
```

The ***sync*** instruction creates a memory barrier (see Section 1.7.1). The set of storage accesses and/or data transfers that is ordered by the memory barrier depends on the contents of the L and SC fields as follows.

SC≠0

- **SC=1 (“store not caching inhibited sync”)**
  The memory barrier provides an ordering function for the storage accesses caused by 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 not Caching Inhibited. The applicable pairs are all pairs a_i,b_j of such storage accesses.
- **SC=2 (“store caching inhibited sync”)**
  The memory barrier provides an ordering function for the storage accesses caused by 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 Caching Inhibited. The applicable pairs are all pairs a_i,b_j of such storage accesses.
- **SC=3 (“store sync”)**
  The memory barrier provides an ordering function for the storage accesses caused by Store and dcbz instructions that are executed by the processor executing the sync instruction. The applicable pairs are all pairs a_i,b_j of such storage accesses.

SC=0

- **L=0 (“heavyweight sync”)**
  The memory barrier provides an ordering function for the storage accesses and data transfers associated with all instructions that are executed by the processor executing the sync instruction with the exception of ***dcbfps*** and ***dcbstps***. The applicable pairs are all pairs a_i,b_j of such storage accesses and data transfers in which b_j is a data access or data transfer, except that if a_i is the storage access caused by an ***icbi*** instruction then b_j may be performed with respect to the processor executing the ***sync*** instruction before a_i is performed with respect to that processor.
- **L=1 (“lightweight sync”)**
  The memory barrier provides an ordering function for the storage accesses caused by *Load*, *Store*, and ***dcbz*** instructions that are executed by the processor executing the sync instruction and for which the specified storage location is in storage that is Memory Coherence Required and is neither Write Through Required nor Caching Inhibited. The applicable pairs are all pairs a_i,b_j of such storage accesses except those in which a_i is 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 6.9.2 of Book III, as are additional properties of the ***sync*** instruction with L=2.
- **L=4 (“persistent heavyweight sync”)**The ordering done by the memory barrier is the same as for ***sync*** with L=0, but extended by adding accesses caused by ***dcbfps*** and ***dcbstps*** to both the set A and the set B of the barrier. In addition, the memory barrier ensures that all stores for which the modifications are written to persistent storage by preceding ***dcbfps*** and ***dcbstps*** instructions have updated persistent storage before any data access or data transfer caused by subsequent instructions is initiated.
- **L=5 (“persistent lightweight sync”)**The ordering done by the memory barrier is the same as for ***sync*** with L=1, but extended by adding accesses caused by ***dcbfps*** and ***dcbstps*** to both the set A and the set B of the barrier. In addition, the memory barrier ensures that all stores for which the modifications are written to persistent storage by preceding ***dcbfps*** and ***dcbstps*** instructions have updated persistent storage before any store in set B updates persistent storage.

The ordering done by the memory barrier is cumulative (regardless of the L and SC values).

If L=0 or L=4 (or L=2), the ***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 991).

  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 1.8 contains a detailed description of how to modify instructions such that a well-defined result is obtained.

The L values 3, 6, and 7 are reserved.

The ***sync*** instruction may complete before storage accesses associated with instructions preceding the ***sync*** instruction have been performed.

Figure 4.4 shows the valid combinations of SC and L values. Instructions that use any of these combinations will execute correctly on processors that comply with versions of the architecture that precede Version 3.1 (in which versions the L field is two bits long, the SC field does not exist, and bits 8 and 14:15 of the ***sync*** instruction are reserved) and on processors that comply with Version 3.1 and subsequent versions of the architecture. If any other combination is used, the instruction form is invalid.

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. Throughout Books I-III, references to the L field imply SC=0 unless otherwise stated or obvious from context. The SC field is mentioned explicitly, or the appropriate extended mnemonics are used, when non-zero SC values are intended. Some programming examples and recommendations assume a programming model that does not include the store-specific variants of ***sync***. Improved performance may be achieved through the use of store-specific memory barriers in some cases.

Programming Note

sync serves as both a basic and an extended mnemonic. The Assembler will recognize a ***sync*** mnemonic with two operands as the basic form, and a ***sync*** mnemonic with one operand or with no operand as an extended form. In the extended form with one operand the SC operand is omitted and assumed to be 0. In the extended form with no operand the L and SC operands are omitted and assumed to be 0.

Architecture Note

In case it proves desirable in a future version of the architecture to add a variant of the store/store memory barriers that also orders writes to persistent storage caused by ***dcbstps*** and ***dcbfps*** instructions, the L,SC pair (5,1) should be reserved for this purpose.

Special Registers Altered:

None

Extended Mnemonics:

Extended Mnemonics for *Synchronize*:

Extended mnemonic:	Equivalent to:
`sync`	`sync 0,0`
`sync L`	`sync L,0`
`hwsync`	`sync 0,0`
`lwsync`	`sync 1,0`
`ptesync`	`sync 2,0`
`phwsync`	`sync 4,0`
`plwsync`	`sync 5,0`
`stncisync`	`sync 1,1`
`stcisync`	`sync 0,2`
`stsync`	`sync 0,3`

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 1046.

The memory barrier created by a ***sync*** instruction with L=1 (or with SC≠0) 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 with SC≠0, or the ***eieio*** instruction, if any of these is sufficient; [] otherwise ***sync*** with L=0 should be used (or with L=4 or L=5 if accesses to persistent storage need to be ordered). ***sync*** with L=2 should not be used by application programs.

Programming Note

The functions provided by ***sync*** with L=1 and with SC≠0 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=4; see Book III.)

Architecture Note

The L values 3, 6, and 7 may be assigned meanings in some future version of the architecture. It is expected that any new meaning will be such that the functions performed by ***sync*** with L=3 are a subset of the functions performed by ***sync*** with L=2. It is expected that any new meaning will be such that the functions performed by ***sync*** with L=6 are a subset of the functions performed by ***sync*** with L=0. It is expected that any new meaning will be such that the functions performed by ***sync*** with L=7 are a subset of the functions performed by ***sync*** with L=1.

Engineering Note

A ***sync*** instruction with L=3 should be implemented as if L=2. A ***sync*** instruction with L=6 should be implemented as if L=0. A ***sync*** instruction with L=7 should be implemented as if L=1. (See the preceding Architecture Note.)

Engineering Note

Unlike a context synchronizing operation, ***sync*** need not cause prefetched instructions to be discarded.

The memory barrier created by the ***sync*** 0 instruction orders the storage accesses caused by the ***icbi*** instruction except as noted in the description of that instruction (see page 991).

SC	L	Intended barrier for processors that comply with V. 3.1 or later	Intended barrier for processors that comply with V. 3.0C or earlier
1	1	stncisync	lwsync
2	0	stcisync	hwsync
3	0	stsync	hwsync
0	0	hwsync	hwsync
0	1	lwsync	lwsync
0	2	ptesync	ptesync
0	4	phwsync	hwsync*
0	5	plwsync	lwsync*
* depends on details of the bus interface design to have proper persistent storage semantics			

Figure 4.4: Interpretation of the L and SC fields

Enforce In-order Execution of I/O X-form

eieio

31	///	///	///	854	/
0	6	11	16	21	31

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

Programming Note

The ***eieio*** instruction is intended for use in doing memory-mapped I/O). Because loads, and separately stores, to storage that is both Caching Inhibited and Guarded are performed in program order (see Section 1.7.1, “Storage Access Ordering ” on page 973), ***eieio*** is needed for such storage only when loads must be ordered with respect to stores.

For the ***eieio*** instruction, accesses in set 1, a_i and b_j need not be the same kind of access or be to storage having the same storage control attributes. For example, a_i can be a load to Caching Inhibited, Guarded storage, and b_j a store to Write Through Required storage.

If stronger ordering is desired than that provided by ***eieio***, the ***sync*** instruction must be used, with the appropriate value in the L field.

Programming Note

The functions provided by ***eieio*** for its second set are a strict subset of those provided by ***sync*** with L=1.

Engineering Note

See the descriptions of ***tlbie*** and ***tlbsync*** in Book III for additional operations that are ordered by ***eieio***.

4.6.4 Wait Instruction

The wait instruction is used to stop instruction fetching and execution until certain events occur. These events include exceptions (see Section 1.2.1 of Book III), event-based branch exceptions (see Section 1.1), the passage of a specified amount of time, and the modification of a storage location.

Wait X-form

wait WC,PL

31	??	/	WC	///	PL	///	30	/
0	6	8	9	11	14	16	21	31

The ***wait*** instruction causes instruction fetching and execution to be suspended under certain conditions, depending on the values of the WC and PL fields. Instruction fetching and execution are resumed when the events specified by the WC field occur or in the rare case of an implementation-dependent event.

The values of the WC field are as follows.

0 Resume instruction fetching and execution when an exception or event-based branch exception [] occurs.

1 Resume instruction fetching and execution when an exception or event-based branch exception occurs, or when a reservation made by the processor does not exist (see Section 1.7.2.1).

2 Resume instruction fetching and execution when an exception or event-based branch exception occurs, or when the amount of time specified by the PL field has passed.

3 Reserved.

The values of the PL field are as follows.

0b00 A short wait time is specified.

0b01:11 Reserved.

If WC=0, or if WC=1 and a reservation made by the processor exists, or if WC=2 and a value for PL that is not reserved is specified, the following applies.

- Upon completion of the instruction, instruction fetching and execution is suspended.
- Instruction fetching and execution resumes when any of the following conditions are met.
  - An exception or event-based branch exception occurs.
  - WC=1 and a reservation made by the processor does not exist.
  - WC=2 and the specified amount of time has passed.
  - An implementation-dependent event occurs.

Programming Note

Because the waiting begins when the instruction completes, if the waiting is ended by an exception that causes a change of control flow (interrupt, event-based branch), the SPR that is set to reflect the point in the instruction stream at which the change of control flow occurred (e.g., SRR0 for a Decrementer interrupt) will contain the EA of the instruction following the ***wait*** instruction.

If WC=1 and a reservation made by the processor does not exist, or if WC=2 and a reserved value of PL is specified, the instruction is treated as a no-op.

Programming Note

Bits 6 and 7 of the ***wait*** instruction may be used in some implementations for an implementation-dependent field. Unless the intention is to use the implementation-dependent field, these bits must be coded zero.

Special Registers Altered:

None

Extended Mnemonics:

Extended Mnemonics for *wait*:

Extended mnemonic:	Equivalent to:
`wait`	`wait 0,0`
`wait WC`	`wait WC,0`
`waitrsv`	`wait 1,0`
`pause_short`	`wait 2,0`

Except in this section, references to "***wait***" in Books I-III include all defined forms of ***wait*** unless otherwise stated or obvious from context.

Programming Note

wait serves as both a basic and an extended mnemonic. The Assembler will recognize a ***wait*** mnemonic with two operands as the basic form and a ***wait*** mnemonic with one operand or with no operand as an extended form. In the extended form with one operand the PL operand is omitted and assumed to be 0. In the extended form with no operand the WC and PL operands are omitted and assumed to be 0.

Programming Note

The ***wait*** instruction frees computational resources which might be allocated to another program or converted into power savings.

Programming Note

Since exceptions corresponding to system-caused interrupts (see Section 7.4 of Book III) may occur at any time, including immediately prior to the ***wait*** instruction, applications should not depend on them to cause ***wait*** to resume. In order to ensure timely resumption, therefore, applications should execute ***wait*** only in order to suspend processing until an event-based branch exception or loss of reservation occurs or a specified amount of time has passed.

Also, since exceptions corresponding to interrupts can cause ***wait*** to resume at any time without any EBB exception or loss of reservation having occurred, and in consideration of the possibility of resuming because of an implementation-dependent event, programs that execute ***wait*** should check that the expected condition has actually occurred after the ***wait*** instruction completes. If the expected condition has not occurred, ***wait*** should be re-executed. An example code usage is shown below.

```
while (¬expected condition), wait
```

Programming Note

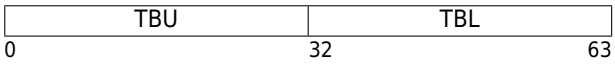
Applications that execute ***wait*** in order to suspend processing until an external event-based branch exception occurs (see Section 6.2) should enable external event-based branch exceptions (by setting $BESCR_{EE}$=1) and disable event-based branches (by setting $BESCR_{GE}$=0) before executing ***wait***. If $BESCR_{GE}$=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.

Engineering Note

The short wait is expected to be somewhere in the range from 10 to 70ns.

Chapter 5. Time Base

The Time Base (TB) is a 64-bit register (see Figure 5.1) containing a 64-bit unsigned integer that is incremented periodically as described below.



Field	Description
TBU	Upper 32 bits of Time Base
TBL	Lower 32 bits of Time Base

Figure 5.1: Time Base

The Time Base monotonically increments until its value becomes 0xFFFF_FFFF_FFFF_FFFF ($2^{64} - 1$); at the next increment its value becomes 0x0000_0000_0000_0000. There is no interrupt or other indication when this occurs.

The suggested frequency at which the time base increments is 512 MHz, however, variation from this rate is allowed provided the following requirements are met.

- The contents of the Time Base differ by no more than +/- four counts from what they would be if they incremented at the required frequency.
- Bit 63 of the Time Base is set to 1 between 30% and 70% of the time over any time interval of at least 16 counts.

The Power ISA does not specify a relationship between the frequency at which the Time Base is updated and other frequencies, such as the CPU clock or bus clock. The Time Base update frequency is not required to be constant. What *is* required, so that system software can keep time of day and operate interval timers, is one of the following.

- The system provides an (implementation-dependent) interrupt to software whenever the update frequency of the Time Base changes, and a means to determine what the current update frequency is.
- The update frequency of the Time Base is under the control of the system software.

Engineering Note

See Book III for additional requirements related to secure systems.

Programming Note

If the operating system initializes the Time Base on power-on to some reasonable value and the update frequency of the Time Base is constant, the Time Base can be used as a source of values that increase at a constant rate, such as for time stamps in trace entries.

Even if the update frequency is not constant, values read from the Time Base are monotonically increasing (except when the Time Base wraps from 2^{64}-1 to 0). If a trace entry is recorded each time the update frequency changes, the sequence of Time Base values can be post-processed to become actual time values.

Successive readings of the Time Base may return identical values.

5.1 Time Base Instructions

Move From Time Base XFX-form

mftb RT,TBR [Phased-Out]

31	RT	tbr	371	/
0	6	11	21	31

This instruction behaves as if it were an ***mfspr*** instruction; see the ***mfspr*** instruction description in Section 3.3.19 of Book I.

Special Registers Altered:

None

Extended Mnemonics:

Extended Mnemonics for *Move From Time Base*:

Extended mnemonic:		Equivalent to:	
mftb	RT	mftb	RT,268
mftb	RT	mfspr	RT,268
mftbu	RT	mftbu	RT,269
mftbu	RT	mfspr	RT,269

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).

Architecture Note

Assemblers targeting Architecture Version 2.03 and later should generate an ***mfspr*** instruction for the ***mftb*** and ***mftbu*** extended mnemonics, and should generate an ***mfspr*** instruction for the ***mftb*** mnemonic specifying TBR 268 or 269. (A basic ***mftb*** mnemonic specifying a TBR value other than 268 or 269 should be considered a programming error.)

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.)

Architecture Note

The ***mftb*** instruction can be removed from the architecture when software that will run on the new implementations no longer uses it. (Because the ***mftb*** instruction will then become an illegal instruction, it will cause an the Illegal Instruction error handler to be invoked and can be emulated by the operating system. However, such emulation may be unacceptable to some software because it reduces significantly the timeliness of the Time Base value returned to the requesting program.)

Engineering Note

The extended opcode for ***mftb*** differs from that of ***mfspr*** by only one bit. This facilitates treating ***mftb*** as if it were ***mfspr***.

Programming Note

Since the update frequency of the Time Base is implementation-dependent, the algorithm for converting the current value in the Time Base to time of day is also implementation-dependent.

As an example, assume that the Time Base increments at the constant rate of 512 MHz. (Note, however, that programs should allow for the possibility that some implementations may not increment the least-significant 4 bits of the Time Base at a constant rate.) What is wanted is the pair of 32-bit values comprising a POSIX standard clock:[1] 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 *ticks_per_sec* contains the value 512,000,000, which is the number of times the Time Base is updated each second.
- The integer constant *ns_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 ***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
divdu   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     Rx,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 *ticks_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.

[1] Described in POSIX Draft Standard P1003.4/D12, *Draft Standard for Information Technology – Portable Operating System Interface (POSIX) – Part 1: System Application Program Interface (API) - Amendment 1: Real-time Extension* [*C Language*]. Institute of Electrical and Electronics Engineers, Inc., Feb. 1992.

Chapter 6. Event-Based Branch Facility

6.1 Event-Based Branch Overview

The Event-Based Branch facility allows application programs to enable hardware to change the effective address of the next instruction to be executed when certain events occur to an effective address specified by the program.

The operation of the Event-Based Branch facility is summarized as follows:

- The Event-Based Branch facility is available only when the system software has made it available. See Section 11.5 of Book III for additional information.
- When the Event-Based Branch facility is available, event-based branches are caused by event-based exceptions. Event-based exceptions can be enabled to occur by setting bits in the BESCR.
- When an event-based exception occurs, the bit in the BESCR control field corresponding to the event-based exception is set to 0 and the bit in the Event Status field in the BESCR corresponding to the event-based exception is set to 1.
- If the global enable bit in the BESCR is set to 1 when any of the bits in the status field are set to 1 (i.e., when an event-based exception exists), an event-based branch occurs.
- The event-based branch causes the following to occur.
  - The global enable bit is set to 0.
  - Bits 0:61 of the EBBRR are set to the effective address of the instruction that would have attempted to execute next if the event-based branch did not occur.
  - Instruction fetch and execution continues at the effective address contained in the EBBHR.
- The event-based branch handler performs the necessary processing in response to the event, and then executes an ***rfebb*** instruction in order to resume execution at the instruction at the address indicated in the EBBRR. See the Programming Notes in Section 6.3 for an example sequence of operations of the event-based branch handler.

Additional information about the Event-Based Branch facility is given in Section 4.4 of Book III.

Programming Note

Since system software controls the availability of the Event-Based Branch facility (see Section 11.5 of Book III), an interface must be provided that enables applications to request access to the facility and determine when it is available.

Programming Note

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 $BESCR_{PME\ PMEO}$ = 1 0, and also sets $MMCR0_{PMAE\ PMAO}$ = 1 0. See Section 11.4.4 of Book III for the description of MMCR0.
- Software sets the GE bit in the BESCR to enable event-based branches.

Initializing the Event-Based Branch facility for External EBB exceptions follows a similar process except that EBB exceptons for these facilities are controlled by different bits in the BESCR.

6.2 Event-Based Branch Registers

6.2.1 Branch Event Status and Control Register

The Branch Event Status and Control Register (BESCR) is a 64-bit register that contains control and status information about the Event-Based Branch facility.

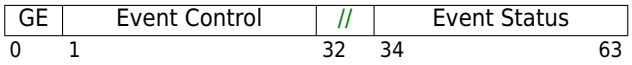


Figure 6.1: Branch Event Status and Control Register (BESCR)

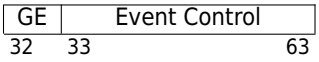


Figure 6.2: Branch Event Status and Control Register Upper (BESCRU)

System software controls whether or not event-based branches occur regardless of the contents of the BESCR. See Section 11.4.4 of Book III and Section 7.2.13 of Book III.

The entire BESCR can be [] 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 SPR 800, 802, or 806, the contents of the BESCR are returned. When mfspr indicates SPR 801 or 803, the contents of the BESCRU are returned.

Programming Note

Event-based branch handlers typically reset event status bits upon entry, and enable event enable bits after processing an event. Execution of ***rfebb*** then re-enables the GE bit so that additional event-based branches can occur.

Bit(s) Definition

0 **Global Enable** (GE)

0 Event-based branches are disabled

1 Event-based branches are enabled.

When an event-based branch occurs, GE is set to 0 and is not altered by hardware until ***rfebb*** 1 is executed or software sets GE=1 and another event-based branch occurs.

1:31 **Event Control**

1:29 Reserved

30 **External Event-Based Exception Enable** (EE)

0 External event-based (EBB) exceptions are disabled.

1 External EBB exceptions are enabled until an external event-based exception occurs, at which time:

– EE is set to 0

– EEO is set to 1

External event-based exceptions exist in any privilege state when an external EBB input from the platform is active. See the system documentation for information about the external EBB input.

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 11 of Book III for information about Performance Monitor event-based exceptions and about the effects of this bit on the Performance Monitor.

Programming Note

Performance Monitor event-based exceptions can only occur in problem state. See Section 11.2 of Book III.

32:33 **Reserved**

Programming Note

Bits 32:33 must contain 0b00 when ***rfebb*** is executed; otherwise the instruction is treated as if the instruction form were invalid.

Architecture Note

Bits 32:33 will be among the last to be assigned a meaning. They were the TS (Transaction State) field in earlier versions of the architecture.

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.

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.

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 11 of Book III for information about Performance Monitor event-based exceptions and about the effects of this bit on the Performance Monitor.

Programming Note

After handling an event-based branch, software should set the "exception occurred" bit(s) corresponding to the event-based exception(s) that have occurred to 0. See the Programming Notes in Section 6.3 for additional information.

6.2.2 Event-Based Branch Handler Register

The Event-Based Branch Handler Register (EBBHR) is a 64-bit register 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.

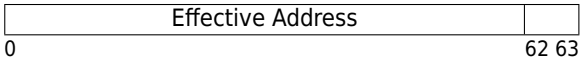


Figure 6.3: Event-Based Branch Handler Register (EBBHR)

Programming Note

The EBBHR can be used by software as a scratchpad register after entry into an event-based branch handler, provided that its contents are restored prior to executing ***rfebb*** 1. An example of such usage is as follows. In the example, SPRG3 is used to contain a pointer to a storage area where private application data may be saved, however, refer to the applicable operating system documentation to determine if an alternate register or storage area should be used.

```
E:mtspr EBBHR, r1      // Save r1 in EBBHR
  mfspr r1, SPRG3      // Move SPRG3 to r1
  std r2,offset1(r1)   // Store r2
  mfspr EBBHR,r2       // Copy original contents
                       // of r1 to r2
  std r2,offset2(r1)   // save original r1
  ...                  // Store rest of state
  ...                  // Process event(s)
  ...                  // Restore all state                    // e
  r2 = &E              // Generate original value
                       // of EBBHR in r2
  mtspr EBBHR,r2       // Restore EBBHR
  ld r2,offset1(r1)    // restore r2
  ld r1,offset2(r1)    // restore r1
  rfebb 1              // Return from handler
```

6.2.3 Event-Based Branch Return Register

The Event-Based Branch Return Register (EBBRR) is a 64-bit register that contains the 62 most significant bits of an instruction effective address as specified below.

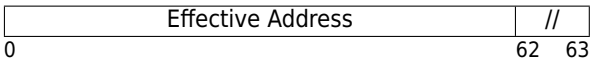


Figure 6.4: Event-Based Branch Return Register (EBBRR)

When an event-based branch occurs, bits 0:61 of the EBBRR are set to the effective address of the instruction that would have attempted to execute next if the event-based branch did not occur.

Bits 62:63 are reserved.

6.3 Event-Based Branch Instructions

Return from Event-Based Branch XL-form

rfebb S

19	///	///	///	S	146	/
0	6	11	16	20	21	31

```
BESCR_GE ← S
NIA ←_iea EBBRR_0:61 || 0b00
```

$BESCR_{GE}$ is set to S.

If there are no pending event-based exceptions, then the next instruction is fetched from the address $EBBRR_{0:61}$ || 0b00 (when MSR_{SF}=1) or $^{32}0$ || $EBBRR_{32:61}$ || 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.

If $BESCR_{32:33} \neq 0b00$ the instruction is treated as if the instruction form were invalid.

See Section 4.4 of Book III for additional information about this instruction.

Special Registers Altered:

Register	*Field(s)*
BESCR	
MSR	(See Book III)

Extended Mnemonics:

Extended Mnemonics for *Return from Event-Based Branch*:

Extended mnemonic:	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

When an event-based branch occurs, the event-based branch handler can execute the following sequence of operations. This sequence of operations assumes that the handler routine has access to a stack or other area in memory in which state information from the main program can be stored. Note also that in this example, the handler entry point is labeled “E,” r1 and r2 are used as scratch registers, and both external EBB and Performance Monitor EBB exceptions are enabled.

```
E:Save state                  // This is the entry point
  mfspr r1, BESCR             // Check event status
  if r1[63]=1, then
    Process PM exception
    r2 ← 0x0000 0000 0000 0001
    mtspr BESCRR, r2          // Reset PMEO status bit
    r2 ← 0x0000 0001 0000 0000
    mtspr BESCRS, r1          // Re-enable PM exceptions
    // Note: The PMAE bit of MMCR0 must also be enabled. See Book III.

  if r1[62]=1, then
    Process external exception
    r2 ← 0x0000 0000 0000 0002
    mtspr BESCRR, r2          // Reset EEO status bit
    r2 ← 0x0000 0002 0000 0000
    ...                       // De-activate external EBB input from platform
    mtspr BESCRS, r1          // Re-enable external EBB exceptions
  // .  .  .
  // Other exceptions are processed similarly.
  // .  .  .
  Restore state
  rfebb 1            // return & global enable
```

Note that before resetting the $BESCR_{EEO}$, the external EBB input from the platform should be deactivated, and additional operations to manage the external EBB input may be required. See the system documentation for details.

In the above sequence, if other exceptions occur after they are enabled, another event-based branch will occur immediately after ***rfebb*** is executed.

Chapter 7. Branch History Rolling Buffer

The Branch History Rolling Buffer (BHRB) is a buffer containing an implementation-dependent number of entries, referred to as BHRB Entries (BHRBEs), that contain information related to branches that have been taken. Entries are numbered from 0 through n, where n is implementation-dependent but no more than 1023. Entry 0 is the most-recently written entry. The BHRB is read by means of the ***mfbhrbe*** instruction.

System software typically controls the availability of the BHRB as well as the number of entries that it contains. If the BHRB is accessed when it is unavailable, the system facility unavailable error handler is invoked.

Various events or actions by the system software may result in the BHRB occasionally being cleared. If BHRB entries are read after this has occurred, 0s will be returned. See the description of the ***mfbhrbe*** instruction for additional information.

The BHRB is typically used in conjunction with Performance Monitor event-based branches. (See Chapter 6 of Book II.) When used in conjunction with this facility, $BESCR_{PME}$ is set to 1 to enable Performance Monitor event-based exceptions, and Performance Monitor alerts are enabled to enable the writing of BHRB entries. When a Performance Monitor alert occurs, Performance Monitor alerts are disabled, BHRB entries are no longer written, and an event-based branch occurs. (See Chapter 11 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 11.4.7 of Book III.

The following paragraphs describe the entries written into the BHRB for various types of *Branch* instructions for which the branch was taken. In some circumstances, however, the hardware may be unable to make the entry even though the following paragraphs require it. In such cases, the hardware sets the EA field to 0, and indicates any missed entries using the T and P fields. (See Section 7.1.)

When an I-form or B-form *Branch* instruction is entered into the BHRB, bits 0:61 of the effective address of the *Branch* instruction are written into the next available entry, except that the entry may or may not be written in the following cases.

- The effective address of the branch target exceeds the effective address of the *Branch* instruction by 4.
- The instruction is a B-form *Branch*, the effective address of the branch target exceeds the effective address of the *Branch* instruction by 8, and the instruction immediately following the *Branch* instruction is not another *Branch* instruction.

The determination of whether the effective address of the branch target exceeds the effective address of the Branch instruction by 4 or 8 is made modulo 2^{64}.

Programming Note

The cases described above, for which the BHRBE need not be written, are cases for which some implementations may optimize the execution of the Branch instruction (first case) or of the Branch instruction and the following instruction (second case) in a manner that makes writing the BHRBE difficult. Such implementations may provide a means by which system software can disable these optimizations, thereby ensuring that the corresponding BHRBEs are written normally.

When an XL-form *Branch* instruction is entered into the BHRB, bits 0:61 of the effective address of the *Branch* instruction are written into the next available entry if allowed by the filtering mode; subsequently, bits 0:61 of the effective address of the branch target are written into the following entry.

7.1 Branch History Rolling Buffer Entry Format

Branch History Rolling Buffer Entries (BHRBEs) have the following format.

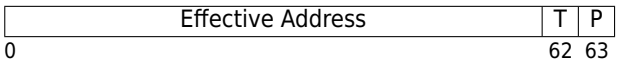


Figure 7.1: Branch History Rolling Buffer Entry

Bit(s) Definition

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.

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.

Value	Meaning
00	This entry either is not implemented or has been cleared. There are no valid entries beyond the current entry.
01-11	Reserved.

7.2 Branch History Rolling Buffer Instructions

The Branch History Rolling Buffer instructions enable application programs to clear and read the BHRB. The availability of these instructions is controlled by the system software. (See Chapter 11 of Book III.) When an attempt is made to execute these instructions when they are unavailable, the system facility unavailable error handler is invoked.

Clear BHRB X-form

clrbhrb

31	///	///	///	430	/
0	6	11	16	21	31

```
for n = 0 to (number_of_BHRBEs implemented - 1)
  BHRB(n) ← 0
```

All BHRB entries are set to 0s.

Special Registers Altered:

None

Move From Branch History Rolling Buffer Entry XFX-form

mfbhrbe RT,BHRBE

31	RT	BHRBE	302	/
0	6	11	21	31

```
n ← BHRBE0:9
If n < number of BHRBEs implemented then
 RT ← BHRBE(n)
else
 RT ← 64 0
```

The BHRBE field denotes an entry in the BHRB. If the designated entry is within the range of BHRB entries implemented and Performance Monitor alterts are disable (see Section 11.5 of Book III), the contents of the designated BHRB entry are placed into register RT; otherwise, zero is 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 6 of Book II) has been executed
- a context synchronizing event (see Section 1.5 of Book III) other than ***isync*** (see Section 4.6.1 of Book II) has occurred.

Special Registers Altered:

None

Programming Note

In order to read all the BHRB entries containing information about taken branches, software should read the entries starting from entry number 0 and continuing until an entry containing all 0s is read or until all implemented BHRB entries have been read.

Since the number of BHRB entries may decrease or the BHRB may be cleared at any time, if a given entry, m, is read as not containing all 0s and is read again subsequently, the subsequent read may return all 0s even though the program has not executed ***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.

dcbtds	RA,RB,TH	(equivalent to: dcbt for TH values of 0b00000 or 0b01000 – 0b01111; other TH values are invalid.)
dcbtt	RA,RB	(equivalent to: dcbt for TH value of 0b10000)
dcbna	RA,RB	(equivalent to: dcbt for TH value of 0b10001)
dcbtstds	RA,RB,TH	(equivalent to: dcbtst for TH values of 0b00000 or 0b01000 – 0b01111; other TH values are invalid.)
dcbtstt	RA,RB	(equivalent to: dcbtst for TH value of 0b10000)

A.2 Data Cache Block Flush Mnemonics

The L field in the *Data Cache Block Flush* instruction controls the scope of the flush function performed by the instruction, or the scope of the store function when L=6. Extended mnemonics are provided that represent the L value in the mnemonic rather than requiring it to be coded as a numeric operand.

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.

dcbf RA,RB	(equivalent to: dcbf RA,RB,0)
dcbfl RA,RB	(equivalent to: dcbf RA,RB,1)
dcbflp RA,RB	(equivalent to: dcbf RA,RB,3)
dcbfps RA,RB	(equivalent to: dcbf RA,RB,4)
dcbstps RA,RB	(equivalent to: dcbf RA,RB,6)

A.3 Or Mnemonics

The three register fields in the ***or*** instruction can be used to specify a hint indicating how the processor should handle stores caused by previous *Store* or ***dcbz*** instructions. An extended mnemonic is supported that represents the operand values in the mnemonic rather than requiring them to be coded as numeric operands.

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.

Note: ***lbarx***, ***lharx***, ***lwarx***, ***ldarx***, and ***lqarx*** 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 operands 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: lharx RT,RA,RB,0)
lwarx RT,RA,RB	(equivalent to: lwarx RT,RA,RB,0)
ldarx RT,RA,RB	(equivalent to: ldarx RT,RA,RB,0)
lqarx RT,RA,RB	(equivalent to: lqarx RT,RA,RB,0)

A.5 Synchronize Mnemonics

The L and SC fields in the *Synchronize* instruction control the scope of the synchronization function performed by the instruction. Extended mnemonics are provided that

represent the L and SC values in the mnemonic rather than requiring them to be coded as numeric operands.

Note: ***sync*** serves as both a basic and an extended mnemonic. The Assembler will recognize a ***sync*** mnemonic with two operands as the basic form, and a ***sync*** mnemonic with one operand or with no operand as an extended form. In the extended form with one operand the SC operand is omitted and assumed to be 0. In the extended form with no operand the L and SC operands are omitted and assumed to be 0.

sync	(equivalent to: sync 0,0)
sync x	(equivalent to: sync x,0)
hwsync	(equivalent to: sync 0,0)
lwsync	(equivalent to: sync 1,0)
ptesync	(equivalent to: sync 2,0)
phwsync	(equivalent to: sync 4,0)
plwsync	(equivalent to: sync 5,0)
stncisync	(equivalent to: sync 1,1)
stcisync	(equivalent to: sync 0,2)
stsync	(equivalent to: sync 0,3)

A.6 Wait Mnemonics

The WC field in the ***wait*** instruction determines the conditions under which instruction execution resumes. Extended mnemonics are provided that represent the WC and PL values in the mnemonic rather than requiring them to be coded as numeric operands.

Note: ***wait*** serves as both a basic and an extended mnemonic. The Assembler will recognize a ***wait*** mnemonic with two operands as the basic form and a ***wait*** mnemonic with one operand or with no operand as an extended form. In the extended form with one operand the PL operand is omitted and assumed to be 0. In the extended form with no operand the WC and PL operands are omitted and assumed to be 0.

wait	(equivalent to: wait 0,0)
[]	
waitrsv	(equivalent to: wait 1,0)
pause_short	wait 2,0

A.7 Move To/From Time Base Mnemonics

The tbr field in the *Move From Time Base* instruction specifies whether the instruction reads the entire Time Base or only the high-order half of the Time Base.

mftb Rx	(equivalent to:	mftb Rx,268)
	or:	mfspr Rx,268
mftbu Rx	(equivalent to:	mftb Rx,269)
	or:	mfspr Rx,269

A.8 Return From Event-Based Branch Mnemonic

The S field in the *Return from Event-Based Branch* instruction specifies the value to which the instruction sets the GE field in the BESCR. Extended mnemonics are provided that represent the S value in the mnemonic rather than requiring it to be coded as a numeric operand.

rfebb	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 reserved
     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 reserved
     bne-    loop     #loop if lost reservation
```

Note:

1. The sequence given above can be changed to perform another Boolean operation atomically on a word in storage, simply by changing the ***and*** instruction to the desired Boolean instruction (***or***, ***xor***, etc.).

Test and Set

This version of the “Test and Set” primitive atomically loads a word from storage, sets the word in storage to a nonzero value if the value loaded is zero, and sets the EQ bit of CR Field 0 to indicate whether the value loaded is zero.

In this example it is assumed that the address of the word to be tested is in GPR 3, the new value (nonzero) is in GPR 4, and the old value is returned in GPR 5.

```
loop:
      lwarx   r5,0,r3 #load and reserve
      cmpwi   r5,0    #done if word not equal to 0
      bne-    exit
      stwcx.  r4,0,r3 #try to store non-0
      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 reserved
      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 5.

```
lwz     r4,0(r3) #load shared data
cmpw    r4,r4    #set CR0 to ''equal''
bne-    $-8      #branch never taken
stw     r7,0(r5) #store other shared data
```

An alternative is to use a technique similar to that described in Section B.2.1.2, by causing the ***stw*** to depend on the value returned by the ***lwz*** and omitting the ***cmpw*** and ***bne-***. The dependency could be created by ANDing the value returned by the ***lwz*** with zero and then adding the result to the value to be stored by the ***stw***. If both storage operands are in storage that is neither Write Through Required nor Caching Inhibited, another alternative is to replace the ***cmpw*** and ***bne-*** with an ***lwsync*** instruction.

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.

```
loop:
      lwarx   r2,0,r3  #get next pointer
      stw     r2,0(r4) #store in new element
      lwsync or sync   #order stw before stwcx
      stwcx.  r4,0,r3  #add new element to list
      bne-    loop     #loop if stwcx. failed
```

In the preceding example, if two list elements have next element pointers in the same reservation granule then, in a multiprocessor, "livelock" can occur. (Livelock is a state in which processors interact in a way such that no processor makes forward progress.)

If it is not possible to allocate list elements such that each element's next element pointer is in a different reservation granule, then livelock can be avoided by using the following, more complicated, sequence.

```
      lwz     r2,0(r3) #get next pointer
loop1:
      mr      r5,r2    #keep a copy
      stw     r2,0(r4) #store in new element
      sync             #order stw before stwcx. and
                       #before lwarx
loop2:
      lwarx   r2,0,r3  #get it again
      cmpw    r2,r5    #loop if changed (someone
      bne-    loop1    #  else progressed)
      stwcx.  r4,0,r3  #add new element to list
      bne-    loop2    #loop if failed
```

In the preceding example, livelock is avoided by the fact that each processor re-executes the ***stw*** only if some other processor has made forward progress.

B.4 Notes

The following notes apply to Section B.1 through Section B.3.

1. To increase the likelihood that forward progress is made, it is important that looping on ***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, [] the "bne- exit" could be changed to "bne- loop". However, in some implementations better performance may be obtained by using an ordinary Load instruction to do the initial checking of the value, as follows.

   ```
   loop:
         lwz     r5,0(r3) #load the word
         cmpwi   r5,0     #loop back if word
         bne-    loop     #  not equal to 0
         lwarx   r5,0,r3  #try again, reserving
         cmpwi   r5,0     #  (likely to succeed)
         bne-    loop
         stwcx.  r4,0,r3  #try to store non-0
         bne-    loop     #loop if lost reservation
   ```

3. In a multiprocessor, livelock is possible if there is a *Store* instruction (or any other instruction that can clear another processor's reservation; see Section 1.7.2.1) between the ***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.

Book III

Power ISA Operating Environment Architecture

Chapter 1. Introduction

1.1 Overview

Chapter 1 of Book I describes computation modes, document conventions, a general systems overview, instruction formats, and storage addressing. This chapter augments that description as necessary for the Power ISA Operating Environment Architecture.

1.2 Document Conventions

The notation and terminology used in Book I apply to this Book also, with the following substitutions.

- For “system alignment error handler” substitute “Alignment interrupt”.
- For “system data storage error handler” substitute “Data Storage interrupt”, “Hypervisor Data Storage interrupt”, or “Data Segment interrupt”, as appropriate.
- For “system error handler” substitute “interrupt”.
- For “system floating-point enabled exception error handler” substitute “Floating-Point Enabled Exception type Program interrupt”.
- For “system illegal instruction error handler” substitute “Hypervisor Emulation Assistance interrupt”.
- For “system instruction storage error handler” substitute “Instruction Storage interrupt”, “Hypervisor Instruction Storage interrupt”, or “Instruction Segment interrupt”, as appropriate.
- For “system privileged instruction error handler” substitute “Privileged Instruction type Program interrupt”.
- For “system service program” substitute “System Call interrupt” or “System Call Vectored interrupt”, as appropriate.
- For “system trap handler” substitute “Trap type Program interrupt”.
- For “system facility unavailable error handler” substitute “Facility Unavailable interrupt” or “Hypervisor Facility Unavailable interrupt.”

1.2.1 Definitions and Notation

The definitions and notation given in Book I and Book II are augmented by the following.

- **Threaded processor, single-threaded processor, thread**

  A threaded processor implements one or more “threads”, where a thread corresponds to the Book I/II concept of “processor”. That is, the definition of “thread” is the same as the Book I definition of “processor”, and “processor” as used in Books I and II can be thought of as either a single-threaded processor or as one thread of a multi-threaded processor. Except where the meaning is clear in context or the number of threads does not matter, the only unqualified uses of “processor” in Book III are in resource names (e.g. Processor Identification Register); such uses should be regarded as meaning “threaded processor”. The threads of a multi-threaded processor typically share certain resources, such as the hardware components that execute certain kinds of instructions (e.g., Fixed-Point instructions), certain caches, the address translation mechanism, and certain hypervisor and ultravisor resources.

- **real page**

  A unit of real storage that is aligned at a boundary that is a multiple of its size. The real page size is 4KB.

- **context of a program**

  The state (e.g., privilege and relocation) in which the program executes. The context is controlled by the contents of certain System Registers, such as the MSR and PTCR, of certain 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 of information derived from tables used in address translation.

- **exception**

  An error, unusual condition, or external signal, that may set a status bit and may or may not cause an interrupt, depending upon whether the corresponding interrupt is enabled.

- **interrupt**

  The act of changing the machine state in response to an exception, as described in 7 "Interrupts" on page 1192.

  - ultravisor interrupt
    An interrupt that forces the thread into ultravisor state by explicitly setting $MSR_{S\ HV\ PR}$ to 0b110 (see Section 4.2.1).
  - hypervisor interrupt
    An interrupt that forces the thread into hypervisor state by explicitly setting $MSR_{HV\ PR}$ to 0b10 and is not an ultravisor interrupt.

    All interrupts explicitly set MSR_{PR} to 0.

- **trap interrupt**

  An interrupt that results from execution of a *Trap* instruction.

- **"must"**

  If software that runs in hypervisor state violates a rule that is stated using the word "must" (e.g., "this field must be set to 0"), and the rule pertains to the contents of a hypervisor resource, to executing an instruction that can be executed only in hypervisor state, or to accessing storage in real addressing mode, the results are undefined, and may include altering resources belonging to other partitions, causing the system to "hang", etc. The same is true for software that runs in ultravisor state and violates a "must" rule pertaining to an ultravisor resource or instruction.

- **hardware**

  Any combination of hard-wired implementation, emulation assist, or interrupt for software assistance. In the last case, the interrupt may be to an architected location or to an implementation-dependent location. Any use of emulation assists or interrupts to implement the architecture is implementation-dependent.

- **ultravisor privileged**

  A term used to describe an instruction or facility that is available when and only when the thread is in ultravisor state.

- **hypervisor privileged**

  A term used to describe an instruction or facility that is available when and only when the thread is in hypervisor state.

  Programming Note

  Because ultravisor state is also a hypervisor state, hypervisor privileged instructions and facilities are also available when the thread is in ultravisor state. (The distinct privilege states in which a hypervisor privileged instruction or facility is available are: hypervisor non-ultravisor state, and ultravisor state.)

- **privileged**

  A term used to describe an instruction or facility that is available when and only when the thread is in privileged state.

  Programming Note

  Because hypervisor state is also a privileged state, privileged instructions and facilities are also available when the thread is in hypervisor state (and when the thread is in ultravisor state). (The distinct privilege states in which a privileged instruction or facility is available are: privileged non-hypervisor state, hypervisor non-ultravisor state, and ultravisor state.)

- **privileged state** and **supervisor mode**

  Used interchangeably to refer to a state in which privileged facilities are available.

- **problem state** and **user mode**

  Used interchangeably to refer to a state in which privileged facilities are not available.

- /, //, ///, ... denotes a field that is reserved in an instruction, in a register, or in an architected storage table.

- ?, ??, ???, ... denotes a field that is implementation-dependent in an instruction, in a register, or in an architected storage table.

1.2.2 Reserved Fields

Book I's description of the handling of reserved bits in System Registers, and of reserved values of defined fields of System Registers, applies also to the SLB. Book I's description of the handling of reserved values of defined fields of System Registers applies also to architected storage tables (e.g., the Page Table).

Software should set reserved fields in the SLB and in architected storage tables to zero, because these fields may be assigned a meaning in some future version of the architecture.

Some fields of certain architected storage tables may be written to automatically by the hardware, e.g., Reference and Change bits in the Page Table. When the hardware writes to such a table, the following rules are obeyed.

- Unless otherwise stated, no defined field other than the one(s) specifically being updated are modified.
- Contents of reserved fields are either preserved or written as zero.

1.2.3 Deviations from the Sequential Execution Model

Additional exceptions to the rule that the thread obeys the sequential execution model, beyond those described in Section 2.2 of Book I and in the bullet defining "program order" in Section 1.1 of Book II, are the following.

- A system-caused or imprecise interrupt may occur. In general, the determination of whether an instruction is required by the sequential execution model is not affected by the potential occurrence of a system-caused or imprecise interrupt. (The determination is affected by the potential occurrence of any other kind of interrupt.)

  The only exceptions to the preceding statement are synchronous Machine Check interrupts, which occur at specific points in the program as if they were instruction-caused and precise, and imprecise Floating-Point Enabled Exception type Program interrupts, which can be forced to occur not later than specific points in the program (see Section 4.6.10 of Book I)

Engineering Note

Instruction-caused precise interrupts must be considered in determining whether an instruction is required by the sequential execution model. However, it is always possible to predict whether they might be caused by any given instruction and thus to determine whether subsequent instructions are sure to be required by the sequential execution model.

- A context-altering instruction is executed (13 "Synchronization Requirements for Context Alterations" on page 1278). 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 10.2 of Book III) need not occur until a subsequent context synchronizing operation has occurred.
- A ***mtdexcr*** or ***mthdexcr*** is executed. The execution behavior change described by aspect 0 may not take effect for an indefinite length of time. The execution behavior change described by aspects 1 through 4 need not take effect until the required subsequent synchronization operation has occurred (Chapter 13 of Book III).

1.2.4 Restricting Out-of-Order Execution

Because some classes of security exploits use side-effects of out-of-order execution to infer behavior of or receive information from programs, it may sometimes be necessary to limit out-of-order execution beyond what's necessary to maintain the appearance of compliance with the sequential execution model. This may include restrictions on the otherwise-permitted deviations from the sequential execution model described in Section 1.2.3 and Section 6.5. The *Or Immediate* instruction described in Section 9.2.1 can be used to create a barrier to out-of-order execution.

1.3 General Systems Overview

The hardware contains the sequencing and processing controls for instruction fetch, instruction execution, and interrupt action. Most implementations also contain data and instruction caches. Instructions that the processing unit can execute fall into the following classes:

- instructions executed in the Branch Facility
- instructions executed in the Fixed-Point Facility
- instructions executed in the Floating-Point Facility
- instructions executed in the Vector Facility

Almost all instructions executed in the Branch Facility, Fixed-Point Facility, Floating-Point Facility, and Vector Facility are nonprivileged and are described in Book I. Book II may describe additional nonprivileged instructions (e.g., Book II describes some nonprivileged instructions for cache management). Instructions related to the privileged state, control of hardware resources, control of the storage hierarchy, and all other privileged instructions are described here or are implementation-dependent.

1.4 Exceptions

The following augments the exceptions defined in Book I that can be caused directly by the execution of an instruction:

- the execution of a floating-point instruction when MSR_{FP}=0 (Floating-Point Unavailable interrupt)
- an attempt to modify a hypervisor resource when the thread is in privileged but non-hypervisor state (see Chapter 2), or an attempt to execute a hypervisor-only instruction (e.g., ***tlbie***) when the thread is in privileged but non-hypervisor state
- an attempt to modify an ultravisor resource when the thread is in privileged but non-ultravisor state (see Chapter 3), or an attempt to execute an ultravisor-only instruction (e.g., ***urfid***, ***msgsndu***, ***msgclru***) when the thread is in privileged but non-ultravisor state
- the execution of a traced instruction (Trace interrupt)

- the execution of a Vector instruction when the vector facility is unavailable (Vector Unavailable interrupt)

1.5 Synchronization

The synchronization described in this section refers to the state of the thread that is performing the synchronization.

1.5.1 Context Synchronization

An instruction or event is *context synchronizing* if it satisfies the requirements listed below. Such instructions and events are collectively called *context synchronizing operations*. The context synchronizing operations are the ***isync*** instruction, the *System Linkage* instructions, the ***mtmsr***[***d***] instructions with L=0, and most interrupts (see Section 7.3).

1. The operation causes instruction dispatching (the issuance of instructions by the instruction fetching mechanism to any instruction execution mechanism) to be halted.
2. The operation is not initiated or, in the case of ***isync***, does not complete, until all instructions that precede the operation have completed to a point at which they have reported all exceptions they will cause.
3. The operation ensures that the instructions that precede the operation will complete execution in the context (privilege, relocation, storage protection, etc.) in which they were initiated, except that the operation has no effect on the context in which the associated Reference and Change bit updates are performed.
4. If the operation directly causes an interrupt (e.g., ***sc*** directly causes a System Call interrupt) or is an interrupt, the operation is not initiated until no exception exists having higher priority than the exception associated with the interrupt (see Section 7.9).
5. The operation ensures that the instructions that follow the operation will be fetched and executed in the context established by the operation. (This requirement dictates that any prefetched instructions be discarded and that any effects and side effects of executing them out-of-order also be discarded, except as described in Section 6.5, “Performing Operations Out-of-Order”.)

Programming Note

A context synchronizing operation is necessarily execution synchronizing; see Section 1.5.2.

Unlike the *Synchronize* instruction, a context synchronizing operation does not affect the order in which storage accesses are performed.

Item 2 permits a choice only for ***isync*** (and [***p***]***hwsync*** and ***ptesync***; see Section 1.5.2) because all other execution synchronizing operations also alter context.

Engineering Note

It is permissible for ***isync***, [***p***]***hwsync***, and ***ptesync*** to be treated like all the other execution synchronizing operations with respect to item 2, and treating them thus may simplify implementation. However, permitting ***isync***, [***p***]***hwsync***, and ***ptesync*** to be initiated before preceding instructions have reported all exceptions they will cause may improve performance (e.g., by increasing the overlap of other operations caused by the ***isync***, [***p***]***hwsync***, or ***ptesync*** instruction with the completion of preceding instructions).

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). [***p***]***hwsync*** and ***ptesync*** are treated like ***isync*** with respect to item 2. The execution synchronizing instructions are [***p***]***hwsync***, ***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 subsections, references to the "operating system" in this document include the hypervisor unless otherwise stated or obvious from context.

2.2 Logical Partitioning Control Register (LPCR)

The contents of the LPCR control a number of aspects of the operation of the thread with respect to a logical partition. Below are shown the bit definitions for the LPCR.

Bit(s) Definition

0:3 **Virtualization Control** (VC)

Controls the virtualization of partition memory for partitions that use HPT translation. This field contains three subfields, VPM, ISL, and KBV. Accesses that are initiated in hypervisor state (i.e., $MSR_{HV\ PR}$=0b10) are performed as if VPM=0 and KBV=0. (ISL applies regardless of privilege.)

0 Reserved

1 **Virtualized Partition Memory** (VPM)

Controls whether VPM mode is enabled when address translation is enabled as specified below.

- 0 VPM mode disabled
- 1 VPM mode enabled

When address translation is disabled, VPM mode is enabled. See Section 6.7.2, "Virtualized Partition Memory (VPM) Mode", and Section 6.7.3.3, "Virtual Real Mode Addressing Mechanism", for additional information on VPM mode.

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 $SLB_{L||LP}$ and $PRTE_{STPS}$ were 0b000. When address translation is disabled, the setting of the ISL bit has no effect. ISL mode has no effect on SLB, TLB, and ERAT entry invalidations caused by ***slbie***, ***slbieg***, ***slbia***, ***slbiag***, ***tlbie***, and ***tlbiel***.

Programming Note

Specifying that L||LP=0b000 in $PATE_{PS}$ has the same effect on address translation when translation is disabled as enabling ISL mode when translation is enabled.

ISL mode is needed when a partition is running with address translation enabled because translation uses the SLB, and the contents of the SLB are controlled by the operating system and should not be modified by the hypervisor. ISL mode is not needed when a partition is running with address translation disabled since Virtual Real Mode address translation uses $PATE_{PS}$, which is not visible to the operating system and is in complete control of the hypervisor.

Any PTE with a base page size of 4K satisfies the page size portion of the match criteria for HPT search.

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 $MSR_{HV||PR} \neq 0b10$, Virtual Page Class Key Storage Protection exceptions that occur on storage operand accesses when VPM=0 cause Hypervisor Data Storage interrupts.

Programming Note

Key-Based Virtualization provides an efficient means for the hypervisor to intercept storage references, e.g. MMIO, that must be emulated. (The corresponding behavior for instruction fetching is not desired.) Virtual Page Class Key Storage Protection exceptions not handled by the hypervisor should be reflected to the operating system at its Data Storage interrupt vector with the hypervisor having set $DSISR_{42}$.

4:8 Reserved

9:11 **Default Prefetch Depth** (DPFD)

The DPFD field is used as the default prefetch depth for data stream prefetching when $DSCR_{DPFD}$=0; see page 993.

12:16 Reserved

17:19 **Power-saving mode Exit Cause Enable (Upper Section)** ($PECE_U$)

17 **Hypervisor Virtualization Exit Enable**

0 When the ***stop*** instruction is executed with $PSSCR_{EC}$=1, Hypervisor Virtualization exceptions are not enabled to cause exit from power-saving mode.
1 When the ***stop*** instruction is executed with $PSSCR_{EC}$=1, Hypervisor Virtualization exceptions are enabled to cause exit from power-saving mode.

18:19 Reserved

20:36 Reserved

37 ***Hypervisor Alternate Interrupt Location*** (HAIL)

Controls the effective address offset, or alternate effective address for System Call Vectored, of the interrupt handler and the relocation mode in which it begins execution for all interrupts received in hypervisor state except those subject to the overrides described below.

0 The interrupt is taken with $MSR_{IR\ DR}$ = 0b00 and no effective address offset or alternate effective address.
1 The interrupt is taken with $MSR_{IR\ DR}$ = 0b11. If the interrupt is not System Call Vectored, an effective address offset of 0xc000_0000_0000_4000 is applied. System Call Vectored uses an alternate effective address of 0xc000_0000_0000_3||LEV||0b0_0000.

The overrides mentioned above are as follows. The list should be read from the top down; the first item matching a given situation applies.

- If the interrupt is received in ultravisor state, the interrupt is taken as if $LPCR_{HAIL}$=0.
- Machine Check, System Reset, and Hypervisor Maintenance interrupts are taken as if $LPCR_{HAIL}$=0.
- If the interrupt occurs when MSR_{HV}=1 and either MSR_{IR}=0 or MSR_{DR}=0, the interrupt is taken as if $LPCR_{HAIL}$=0.

38 ***Interrupt Little-Endian*** (ILE)

The contents of the ILE bit are copied into MSR_{LE} by interrupts that set MSR_{HV} to 0 (see Section 7.5), 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 received in privileged non-hypervisor state 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 Reserved
3 The interrupt is taken with $MSR_{IR\ DR}$ = 0b11. If the interrupt is not System Call Vectored, an effective address offset of

0xc000_0000_0000_4000 is applied. System Call Vectored uses an alternate effective address of 0xc000_0000_0000_3 || LEV || 0b0_0000.

[]

The overrides mentioned above are as follows. The list should be read from the top down; the first item matching a given situation applies.

- []
- If the interrupt occurs when MSR_{IR}=0 or MSR_{DR}=0, the interrupt is taken as if $LPCR_{AIL}$=0.

[]

41 **Use Process Table** (UPRT)

Controls whether Process Tables are used. For a radix-using partition, UPRT must be set to 1; UPRT=0 when HR=1 is an unsupported MMU configuration and causes a Hypervisor Data Storage or Hypervisor Instruction Storage interrupt (see Section 7.5.16 and Section 7.5.17). 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.

42 **Enhanced Virtualization** (EVIRT)

Controls whether Enhanced Virtualization is enabled, as specified below.

0 Enhanced Virtualization is disabled: attempts to execute hypervisor-privileged instructions or access hypervisor resources, or PTCR, DAWRn, DAWRXn, or CIABR when they are ultravisor resources, in privileged non-hypervisor state cause a Privileged Instruction type Program interrupt; attempts to access undefined SPR numbers (using **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 execute hypervisor-privileged instructions or access hypervisor resources, or PTCR, DAWRn, DAWRXn, or CIABR when they are ultravisor resources, in privileged non-hypervisor state cause a Hypervisor Emulation Assistance interrupt; attempts to access undefined SPR numbers (using **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 7.5.18, “Hypervisor Emulation Assistance Interrupt”. It also permits emulation of new SPRs on designs that do not support them in hardware.

All accesses to the reserved noop SPRs (808-811) are always treated as noops, independent of the value of EVIRT.

43 **Host Radix** (HR)

Indicates whether the hypervisor uses Radix Tree translation for the partition, as specified below.

0 hypervisor uses HPT translation for this partition.

1 hypervisor uses Radix Tree translation for this partition.

The hypervisor must program HR to match the Host Radix bit in the Partition Table Entry for the partition indicated by LPIDR. If the values do not match and the thread is not in hypervisor real addressing mode or ultravisor real addressing mode, the results are undefined.

Programming Note

HR is duplicated in the LPCR because there are times such as immediately after a partition swap when it is difficult for hardware to quickly access the PATE.

The translation mode for the hypervisor is the same as the translation mode of the partition the hypervisor is serving. This is necessary for consistent, well-defined behavior when a hypervisor concurrently serves partitions using both translation modes, and it creates a requirement that HR=1 in the PATE for LPID=0 when Radix Tree Translation partitions exist in the system, because of the effLPID construct. The architecture may refer to the translation mode of the hypervisor rather than the HR value for the partition when the relationship to the hypervisor matters.

44 Reserved

45 **Online** (ONL)

0 The PURR and SPURR do not increment.

1 The PURR and SPURR increment.

Programming Note

Typically, the hypervisor sets the ONL bit to 0 when the thread is not in a power saving mode, is not performing useful work, and is available for use. The hypervisor may take the state of the ONL bit into account when making course-grain load balancing and power management decisions.

46 **Large Decrementer** (LD)

0 Large Decrementer mode is not enabled.
1 Large Decrementer mode is enabled.

See Section 8.4 for additional information.

47:51 **Power-saving mode Exit Cause Enable (Lower Section)** ($PECE_L$)

47 **Privileged Doorbell Exit Enable**

0 When the ***stop*** instruction is executed with $PSSCR_{EC}$=1, Directed Privileged Doorbell exceptions are not enabled to cause exit from power-saving mode
1 When the ***stop*** instruction is executed with $PSSCR_{EC}$=1, Directed Privileged Doorbell exceptions are enabled to cause exit from power-saving mode.

48 **Hypervisor Doorbell Exit Enable**

0 When the ***stop*** instruction is executed with $PSSCR_{EC}$=1, Directed Hypervisor Doorbell exceptions are not enabled to cause exit from power-saving mode
1 When the ***stop*** instruction is executed with $PSSCR_{EC}$=1, Directed Hypervisor Doorbell exceptions are enabled to cause exit from power-saving mode.

49 **External Exit Enable**

0 When the ***stop*** instruction is executed with $PSSCR_{EC}$=1, External exceptions are not enabled to cause exit from power-saving mode.
1 When the ***stop*** instruction is executed with $PSSCR_{EC}$=1, External exceptions are enabled to cause exit from power-saving mode.

50 **Decrementer Exit Enable**

0 When the ***stop*** instruction is executed with $PSSCR_{EC}$=1, Decrementer exceptions are not enabled to cause exit from power-saving mode.
1 When the ***stop*** instruction is executed with $PSSCR_{EC}$=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.)

51 **Other Exit Enable**

0 When the ***stop*** instruction is executed with $PSSCR_{EC}$=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 $PSSCR_{EC}$=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 7.5.1 and Section 7.5.2 for additional information about exit from power-saving mode.

52 ***Mediated External Exception Request*** (MER)

0 A Mediated External exception is not requested.
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.

- $LPCR_{MER}$ = 1 and a Mediated External exception exists.
- $LPCR_{MER}$ = 0 and a Mediated External exception does not exist.

A context synchronizing instruction or event that is executed or occurs when $LPCR_{MER}$ = 0 ensures that the exception effects of $LPCR_{MER}$ are consistent with the contents of $LPCR_{MER}$. Otherwise, when an instruction changes the contents of $LPCR_{MER}$, the exception effects of $LPCR_{MER}$ become consistent with the new contents of $LPCR_{MER}$ reasonably soon after the change.

Programming Note

$LPCR_{MER}$ 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 $LPCR_{MER}$, the partition can be interrupted upon enabling external interrupts. Without using $LPCR_{MER}$, 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.

53 ***Guest Translation Shootdown Enable*** (GTSE)

Controls whether the operating system is permitted to use ***tlbie***, ***slbieg***, and ***slbiag*** directly, or must issue a system call to the hypervisor.

0 Guest is not permitted to use ***tlbie***, ***slbieg***, ***slbiag***, ***tlbsync***, and ***slbsync***.

1 Guest is permitted to use ***tlbie***, ***slbieg***, ***slbiag***, ***tlbsync***, and ***slbsync***.

Programming Note

An operating system that uses HPT translation must know whether VPM is active in order to invalidate the translation for a specific page using ***tlbie***[***l***]. See the related Programming Notes in the descriptions of ***tlbie*** and ***tlbiel***.

54 ***Translation Control*** (TC)

0 The secondary Page Table search is enabled.

1 The secondary Page Table search is disabled.

55:58 Reserved

59 **Hypervisor External Interrupt Control** (HEIC)

0 Direct External interrupts can occur in hypervisor state.

1 Direct External interrupts cannot occur in hypervisor state.

Programming Note

By setting HEIC=1, the Hypervisor Interrupt Virtualization handler can prevent External interrupts from occurring during the Hypervisor Virtualization interrupt handler. See Section 7.5.7.1.

Engineering Note

The initial value of the HEIC bit (i.e., the value when software begins execution during initialization) must be 0.

60 **Logical Partitioning Environment Selector** (LPES)

0 External interrupts set the HSRRs, set MSR_{HV} to 1, and leave MSR_{RI} unchanged.

1 External interrupts set the SRRs, set MSR_{RI} to 0, and leave MSR_{HV} unchanged.

Programming Note

LPES = 1 should be used by operating systems not running under a hypervisor, so that external interrupts are directed to the SRRs rather than to the HSRRs.

Programming Note

In versions of the architecture that precede Version 2.07, LPES was a two-bit field, in which the second bit controlled significant aspects of storage accessing and interrupt handling.

61 Reserved

62 **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 7.5.21.

Engineering Note

The initial value of the HVICE bit (i.e., the value when software begins execution during initialization) must be 0.

63 **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 7.5.12 on page 1217.

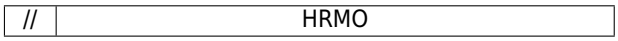
Engineering Note

The initial value of the HDICE bit (i.e., the value when software begins execution during initialization) must be 0.

See Section 7.5 on page 1201 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 2.1 below.



Bits	Name	Description
4:63	HRMO	Real Mode Offset

Figure 2.1: Hypervisor Real Mode Offset Register

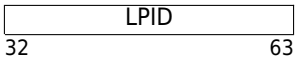
All other fields are reserved.

The supported HRMO values are the non-negative multiples of 2^r, where r is an implementation-dependent value and $12 \leq r \leq 26$.

The contents of the HRMOR affect how some storage accesses are performed as described in Section 6.7.3 on page 1115 and Section 6.7.5 on page 1119.

2.4 Logical Partition Identification Register (LPIDR)

The layout of the Logical Partition Identification Register (LPIDR) is shown in Figure 2.2 below.



Bits	Name	Description
32:63	LPID	Logical Partition Identifier

Figure 2.2: Logical Partition Identification Register

The contents of the LPIDR identify the partition to which the thread is assigned, affecting some aspects of translation. [] The number of LPIDR bits supported is implementation-dependent.

Programming Note

Radix tree translation assigns special meaning to LPID=0, specifically indicating the hypervisor's own partition. When HR=1, LPIDR should not be set to zero except when MSR_{HV}=1.

HPT translation provides special functionality for LPID=0 when HV=1, as described in Section 6.9.3. A partition that uses HPT translation and requires the services of an adjunct should not be assigned LPID=0.

2.5 Processor Compatibility Register (PCR)

The layout of the Processor Compatibility Register (PCR) is shown in Figure 2.3 below.

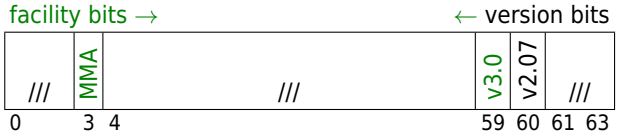


Figure 2.3: Processor Compatibility Register

High-order PCR bits are assigned to control the availability of facilities. Low-order PCR bits are assigned to control the availability of resources that are new in a specified version of the Architecture. The two types of bits can interact. For example, if a facility is created in one version and extended in the next, the high-order bit enables the portion of the facility that was defined in the version of the architecture enabled in the low-order bits.

Each defined bit in the PCR controls whether certain instructions, SPRs, and other related facilities are available in problem state. Except as specified elsewhere in this section, the PCR has no effect on facilities when the thread is not in problem state, or on privileged facilities when the thread is in problem state. Facilities that are made unavailable by the PCR are treated as follows when the thread is in problem state.

- Except for ***hashchk***, ***hashst***, ***mffsl***, ***mffsce***, ***mffsdrn***, ***mffsdrni***, ***mffsrn***, and ***mffsrni***, non-privileged instructions are treated as illegal instructions.
- ***mffsl***, ***mffsce***, ***mffsdrn***, ***mffsdrni***, ***mffsrn***, and ***mffsrni*** perform as if they are an ***mffs*** instruction.
- ***hashchk***, ***hashchkp***, ***hashst***, and ***hashstp*** perform as if they are a no-op instruction.
- SPRs are treated as if they were not defined for the implementation.
- The "reserved SPRs" (see Section 1.3.3 of Book I) are treated as not defined for the implementation.
- Fields in instructions are treated as if they were 0s except for the L field in ***paste.***
- Values of fields in instructions cause the instruction to be treated as an invalid form of the instruction.
- Unless the third item of this list applies, bits in system registers read back 0s for ***mfspr*** and ***mtspr*** operations have no effect on their values, except as described immediately below for bits 44:45 of the XER.

  For bits 44:45 of the XER, two pairs of bits are provided, an "OV32-CA32" bit pair for XER_{OV32} and XER_{CA32} and a "reserved" bit pair for legacy XER bits 44:45 behavior.

  Which bit pair is read by ***mfxer*** is controlled by the PCR. ***mtxer*** writes to both bit pairs, independent of the PCR. ***mcrxr*** reads the "OV32-CA32" bit pair.

  Each bit in the "OV32-CA32" bit pair is implicitly set by instructions that implicitly set their respective XER_{OV} or XER_{CA}, independent of the PCR. The "reserved" bit pair for bits 44:45 of the XER are not altered by these instructions, independent of the PCR.

Programming Note

The "reserved" bit pair does not conform to the usual rules for reading (***mfspr***) reserved bits in registers (see Section 1.3.3 of Book I) because some early implementations used bits 44:45 of the XER for implementation-specific purposes. On these implementations, and on subsequent implementations that implemented versions of the architecture that precede V. 3.0, ***mfxer*** returned the contents of the bits, despite that the bits were defined as reserved.

A defined bit in the PCR may also control whether certain instructions, SPRs, and other related facilities are available in a privileged state (MSR_{PR}=0). Affected facilities will be specifically annotated.

Programming Note

When a bit in a system register is made unavailable by the PCR, ***mtspr*** operations performed on the register in problem state have no effect on the value of the bit regardless of the privilege state in which the register may subsequently be read.

A PCR bit may also determine how an instruction field value is interpreted or may define other behavior as specified in the bit definitions below.

As described in more detail below, the value 1 in a defined bit in the PCR makes the affected resources unavailable and the value 0 makes them available.

The initial state of the PCR is all 0s. In this state, all instructions and facilities supported by the processor are available in all privilege states.

Programming Note

Hypervisors written for a given version of the architecture generally cannot support facilities that are defined in a subsequent version of the architecture if those facilities have new state that would need to be preserved across context switches. (In the absence of such state, support may or may not be possible.) Therefore, hypervisors will set to 1 all PCR bits that are reserved in the given version of the architecture.

The PCR has no effect on the setting of the MSR and [H]SRR1 by interrupts (and of the Count Register by the System Call Vectored interrupt), and by the ***rfscv***, ***rfid***, ***hrfid***, ***urfid*** 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 the BHRB instructions, event-based branch instructions, TAR, DSCR at SPR 3, SIER, MMCR2, the event-based branch instructions, and certain Floating-Point, Vector, and VSX instructions. When any of these facilities are made unavailable in problem state by the PCR, the corresponding [Hypervisor] Facility Unavailable, Floating-Point Unavailable, Vector, or VSX unavailable interrupts do not occur when the facility is accessed in problem state. Note, however, that the PCR does not affect privileged accesses, and thus any Hypervisor Facility Unavailable, Floating-Point Unavailable, Vector unavailable, or VSX unavailable interrupts that are specified to occur as a result of privileged accesses occur regardless of the PCR value.

Programming Note

Because the PCR has no effect on privileged instructions except as specified above, privileged instructions that are available on newer implementations but not available on older implementations will behave differently when the thread is in problem state. On older implementations, a Hypervisor Emulation Assistance interrupt will occur because the instruction is undefined; on newer implementations, a Privileged Instruction type Program interrupt will occur because the instruction is implemented.

In future versions of the architecture, in general the lowest-order reserved bit of the PCR will be used to control the availability of the instructions and related resources that are new in that version of the architecture; the name of the bit will correspond to the previous version of the architecture (i.e., the newest version in which the instructions and related resources were not available).

In these future versions of the architecture, there will be a requirement that if any bit of the low-order defined bits is set to 1 then all higher-order bits of the defined low-order bits must also be set to 1, and the architecture version with which the implementation appears to comply, in problem state, will be the version corresponding to the name of the lowest-order 1 bit in the set of defined low-order PCR bits, or the current architecture version if none of these bits are 1. Also, in general the highest-order reserved bits will be used to control the availability of sets of instructions and related resources having the requirement that their availability be independent of versions of the architecture.

Architecture Note

When a new instruction or related resource is added to the architecture, or an existing instruction or related resource is changed, the following guidelines should be considered in determining whether the availability of the addition or change should be affected by the PCR.

1. In general the PCR should affect only problem state programs. However, if the addition or change would cause old operating systems to run incorrectly on the new design, the PCR should be used to make that addition or change unavailable in privileged but non-hypervisor state.
2. In general the PCR should affect additions or changes that would otherwise permit an old problem state program to run incorrectly on the new design, and should not affect additions or changes that alter only a program's performance (e.g. a new hint field added to an existing instruction).

Engineering Note

Implementation-specific registers (e.g., “HID” registers), rather than the PCR, must be used to control availability of implementation-specific resources and behaviors when such control is needed – i.e., when such a resource or behavior is available in problem state and it would affect problem state program correctness, or when it would prevent old operating systems from running correctly on new designs. These controls must not apply in hypervisor state, and should apply in privileged but non-hypervisor state only if such control is needed in order to permit old operating systems to run correctly on new designs.

The bit definitions for the PCR are shown below.

Bit(s) Definition

0:2 Reserved

3 **Matrix-Multiply Assist** (MMA)

This bit controls the availability of the instructions listed in Table 2.1 on page 1063 for all privilege states.

0 The instructions listed in Table 2.1 are available if the v3.0 bit is also set to 0.

1 The instructions listed in Table 2.1 are not available.

4:58 Reserved

59 **Version 3.0** (v3.0)

When MSR_{PR}=1 (i.e., problem state), this bit controls the availability of the following instructions, facilities, and behaviors that were newly available in problem state in the version of the architecture subsequent to Version 3.0.

- The instructions listed in Table 2.2 on page 1064
- The value 0 in bit 10 of the ***paste.*** instruction. When this value is unavailable, attempt to execute a ***paste.*** instruction with bit 10 set to 0 causes a Hypervisor Emulation Assistance interrupt in all privilege levels. (Although the L field is new, it has behavior more like a new value of a field because its pre-existing value was 1 instead of 0, and there is a need to treat the new value as causing an illegal instruction behavior instead of being ignored.)
- The ability to perform a data transfer from one storage location to another or to another system. When this function is unavailable, attempting to use it in any privilege state causes a DSI or HDSI on the ***paste.*** instruction, setting [H]$DSISR_{60}$, as would occur under Version 3.0 of the architecture. See the latest revision of Version 3.0 of the architecture for details.
- SIER2, SIER3 and MMCR3
- DEXCR and HDEXCR

This bit is encoded as follows.

0 The instructions, behaviors, and facilities listed above are available.

Any word in storage fetched as an instruction having a primary opcode of 0b000001 is treated as the prefix of a prefixed instruction and the next sequential word in storage is treated as the suffix of that prefixed instruction.

Instructions with primary opcode 31 and extended opcodes 658, 690, 722 and 754 are treated as ***hashstp***, ***hashchkp***, ***hashst*** and ***hashchk*** instructions respectively. (Attempt to execute either of the first two instructions causes a Privileged Instruction type Program interrupt.)

1 The instructions, behaviors, and facilities listed above are unavailable.

Any word in storage fetched as an instruction having a primary opcode of 0b000001 is treated as an illegal word instruction.

Instructions with primary opcode 31 and extended opcodes 658, 690, 722 and 754 are treated as no-op instructions.

Because this bit affects whether prefixed instructions are treated as illegal word instructions, it may affect how the Hypervisor Emulation Assist interrupt sets $HSRR1_{34}$ and HEIR.

If this bit is set to 1, then the MMA bit must also be set to 1.

When MSR_{PR}=1 and $MMCR0_{PMCC}$=0b00, this bit controls whether read permission on Group B Performance Monitor registers (i.e., SIER, SIAR, SDAR, and MMCR1 at SPR numbers 768, 780, 781 and 782, respectively) specified in Version 3.0 of the architecture is further conditional on $MMCR0_{PMCCEXT}$ bit or not.

0 ***mfspr*** availability on the mentioned registers is conditional on $MMCR0_{PMCCEXT}$ bit.

1 ***mfspr*** on the mentioned registers is available in problem state without further conditions.

When MSR_{PR}=0 (i.e., privileged state), this bit controls the behavior of instructions with primary opcode 31 and extended opcodes 658, 690, 722 and 754.

0 Instructions with primary opcode 31 and extended opcodes 658, 690, 722 and 754 are treated as ***hashstp***, ***hashchkp***, ***hashst*** and ***hashchk*** instructions respectively.

1 Instructions with primary opcode 31 and extended opcodes 658, 690, 722 and 754 are treated as no-op instructions.

60 **Version 2.07** (v2.07)

When MSR_{PR} =1 (i.e., problem state), this bit controls the availability of the following instructions, facilities, and behaviors that were newly available in problem state in the version of the architecture subsequent to Version 2.07.

- The instructions listed in Table 2.3 on page 1067
- ***scv***

This bit is encoded as follows.

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_{PR} =0 (i.e., privileged or hypervisor-privileged state), this bit controls the availability of the ***mcrxrx*** instruction and which bit pair is read by ***mfxer*** for XER bits 44:45.

0 ***mcrxrx*** is available.

mfxer reads the contents of the "OV32-CA32" bit pair for XER bits 44:45.

1 ***mcrxrx*** is unavailable.

mfxer reads the contents of the "reserved" bit pair for XER bits 44:45.

If this bit is set to 1, then the v3.0 bit must also be set to 1.

61:63 Reserved

Table 2.1: v3.1 instructions controlled by the MMA bit

pmxvbf16ger2	Prefixed Masked VSX Vector bfloat16 GER (Rank-2 Update)
pmxvbf16ger2nn	Prefixed Masked VSX Vector bfloat16 GER (Rank-2 Update) (Negative Multiply, Negative Accumulate)
pmxvbf16ger2np	Prefixed Masked VSX Vector bfloat16 GER (Rank-2 Update) (Negative Multiply, Positive Accumulate)
pmxvbf16ger2pn	Prefixed Masked VSX Vector bfloat16 GER (Rank-2 Update) (Positive Multiply, Negative Accumulate)
pmxvbf16ger2pp	Prefixed Masked VSX Vector bfloat16 GER (Rank-2 Update) (Positive Multiply, Positive Accumulate)
pmxvf16ger2	Prefixed Masked VSX Vector Half-Precision GER (Rank-2 Update)
pmxvf16ger2nn	Prefixed Masked VSX Vector Half-Precision GER (Rank-2 Update) (Negative Multiply, Negative Accumulate)
pmxvf16ger2np	Prefixed Masked VSX Vector Half-Precision GER (Rank-2 Update) (Negative Multiply, Positive Accumulate)
pmxvf16ger2pn	Prefixed Masked VSX Vector Half-Precision GER (Rank-2 Update) (Positive Multiply, Negative Accumulate)
pmxvf16ger2pp	Prefixed Masked VSX Vector Half-Precision GER (Rank-2 Update) (Positive Multiply, Positive Accumulate)
pmxvf32ger	Prefixed Masked VSX Vector Single-Precision GER (Rank-1 Update)
pmxvf32gernn	Prefixed Masked VSX Vector Single-Precision GER (Rank-1 Update) (Negative Multiply, Negative Accumulate)
pmxvf32gernp	Prefixed Masked VSX Vector Single-Precision GER (Rank-1 Update) (Negative Multiply, Positive Accumulate)
pmxvf32gerpn	Prefixed Masked VSX Vector Single-Precision GER (Rank-1 Update) (Positive Multiply, Negative Accumulate)
pmxvf32gerpp	Prefixed Masked VSX Vector Single-Precision GER (Rank-1 Update) (Positive Multiply, Positive Accumulate)
pmxvf64ger	Prefixed Masked VSX Vector Double-Precision GER (Rank-1 Update)
pmxvf64gernn	Prefixed Masked VSX Vector Double-Precision GER (Rank-1 Update) (Negative Multiply, Negative Accumulate)
pmxvf64gernp	Prefixed Masked VSX Vector Double-Precision GER (Rank-1 Update) (Negative Multiply, Positive Accumulate)
pmxvf64gerpn	Prefixed Masked VSX Vector Double-Precision GER (Rank-1 Update) (Positive Multiply, Negative Accumulate)
pmxvf64gerpp	Prefixed Masked VSX Vector Double-Precision GER (Rank-1 Update) (Positive Multiply, Positive Accumulate)
pmxvi16ger2	Prefixed Masked VSX Vector Signed Halfword GER (Rank-2 Update)
pmxvi16ger2pp	Prefixed Masked VSX Vector Signed Halfword GER (Rank-2 Update) (Positive Multiply, Positive Accumulate)
pmxvi16ger2s	Prefixed Masked VSX Vector Signed Halfword GER (Rank-2 Update) with Saturate
pmxvi16ger2spp	Prefixed Masked VSX Vector Signed Halfword GER (Rank-2 Update) with Saturate (Positive Multiply, Positive Accumulate)
pmxvi4ger8	Prefixed Masked VSX Vector Signed Nibble GER (Rank-8 Update)
pmxvi4ger8pp	Prefixed Masked VSX Vector Signed Nibble GER (Rank-8 Update) (Positive Multiply, Positive Accumulate)
pmxvi8ger4	Prefixed Masked VSX Vector Signed/Unsigned Byte GER (Rank-4 Update)
pmxvi8ger4pp	Prefixed Masked VSX Vector Signed/Unsigned Byte GER (Rank-4 Update) (Positive Multiply, Positive Accumulate)
pmxvi8ger4spp	Prefixed Masked VSX Vector Signed/Unsigned Byte GER (Rank-4 Update) with Saturate (Positive Multiply, Positive Accumulate)
xvbf16ger2	VSX Vector bfloat16 GER (Rank-2 Update)
xvbf16ger2nn	VSX Vector bfloat16 GER (Rank-2 Update) (Negative Multiply, Negative Accumulate)
xvbf16ger2np	VSX Vector bfloat16 GER (Rank-2 Update) (Negative Multiply, Positive Accumulate)
xvbf16ger2pn	VSX Vector bfloat16 GER (Rank-2 Update) (Positive Multiply, Negative Accumulate)
xvbf16ger2pp	VSX Vector bfloat16 GER (Rank-2 Update) (Positive Multiply, Positive Accumulate)
xvf16ger2	VSX Vector Half-Precision GER (Rank-2 Update)
xvf16ger2nn	VSX Vector Half-Precision GER (Rank-2 Update) (Negative Multiply, Negative Accumulate)
xvf16ger2np	VSX Vector Half-Precision GER (Rank-2 Update) (Negative Multiply, Positive Accumulate)
xvf16ger2pn	VSX Vector Half-Precision GER (Rank-2 Update) (Positive Multiply, Negative Accumulate)
xvf16ger2pp	VSX Vector Half-Precision GER (Rank-2 Update) (Positive Multiply, Positive Accumulate)
xvf32ger	VSX Vector Single-Precision GER (Rank-1 Update)
xvf32gernn	VSX Vector Single-Precision GER (Rank-1 Update) (Negative Multiply, Negative Accumulate)
xvf32gernp	VSX Vector Single-Precision GER (Rank-1 Update) (Negative Multiply, Positive Accumulate)
xvf32gerpn	VSX Vector Single-Precision GER (Rank-1 Update) (Positive Multiply, Negative Accumulate)
xvf32gerpp	VSX Vector Single-Precision GER (Rank-1 Update) (Positive Multiply, Positive Accumulate)
xvf64ger	VSX Vector Double-Precision GER (Rank-1 Update)
xvf64gernn	VSX Vector Double-Precision GER (Rank-1 Update) (Negative Multiply, Negative Accumulate)
xvf64gernp	VSX Vector Double-Precision GER (Rank-1 Update) (Negative Multiply, Positive Accumulate)
xvf64gerpn	VSX Vector Double-Precision GER (Rank-1 Update) (Positive Multiply, Negative Accumulate)
xvf64gerpp	VSX Vector Double-Precision GER (Rank-1 Update) (Positive Multiply, Positive Accumulate)
xvi16ger2	VSX Vector Signed Halfword GER (Rank-2 Update)
xvi16ger2pp	VSX Vector Signed Halfword GER (Rank-2 Update) (Positive Multiply, Positive Accumulate)
xvi16ger2s	VSX Vector Signed Halfword GER (Rank-2 Update) with Saturate
xvi16ger2spp	VSX Vector Signed Halfword GER (Rank-2 Update) with Saturate (Positive Multiply, Positive Accumulate)
xvi4ger8	VSX Vector Signed Nibble GER (Rank-8 Update)
xvi4ger8pp	VSX Vector Signed Nibble GER (Rank-8 Update) (Positive Multiply, Positive Accumulate)
xvi8ger4	VSX Vector Signed/Unsigned Byte GER (Rank-4 Update)
xvi8ger4pp	VSX Vector Signed/Unsigned Byte GER (Rank-4 Update) (Positive Multiply, Positive Accumulate)
xvi8ger4spp	VSX Vector Signed/Unsigned Byte GER (Rank-4 Update) with Saturate (Positive Multiply, Positive Accumulate)
xxmfacc	VSX Move From ACC
xxmtacc	VSX Move To ACC
xxsetaccz	VSX Set ACC to Zero

Table 2.2: Instructions controlled by the v3.0 bit

brd	Byte-Reverse Doubleword
brh	Byte-Reverse Halfword
brw	Byte-Reverse Word
cfuged	Centrifuge Doubleword
cntlzdm	Count Leading Zeros Doubleword under bit Mask
cnttzdm	Count Trailing Zeros Doubleword under bit Mask
dcffixqq	DFP Convert From Fixed Quadword Quad
dctfixqq	DFP Convert To Fixed Quadword Quad
lxvkq	Load VSX Vector Special Value Quadword
lxvp	Load VSX Vector Paired
lxvpx	Load VSX Vector Paired Indexed
lxvrbx	Load VSX Vector Rightmost Byte Indexed
lxvrdx	Load VSX Vector Rightmost Doubleword Indexed
lxvrhx	Load VSX Vector Rightmost Halfword Indexed
lxvrwx	Load VSX Vector Rightmost Word Indexed
mtvsrbm	Move to VSR Byte Mask
mtvsrbmi	Move To VSR Byte Mask Immediate
mtvsrdm	Move to VSR Doubleword Mask
mtvsrhm	Move to VSR Halfword Mask
mtvsrqm	Move to VSR Quadword Mask
mtvsrwm	Move to VSR Word Mask
paddi	Prefixed Add Immediate
pdepd	Parallel Bits Deposit Doubleword
pextd	Parallel Bits Extract Doubleword
plbz	Prefixed Load Byte & Zero
pld	Prefixed Load Doubleword
plfd	Prefixed Load Floating Double
plfs	Prefixed Load Floating Single
plha	Prefixed Load Halfword Algebraic
plhz	Prefixed Load Halfword & Zero
plq	Prefixed Load Quadword
plwa	Prefixed Load Word Algebraic
plwz	Prefixed Load Word & Zero
plxsd	Prefixed Load VSX Scalar Doubleword
plxssp	Prefixed Load VSX Scalar Single
plxv	Prefixed Load VSX Vector
plxvp	Prefixed Load VSX Vector Paired
pnop	Prefixed No-Operation
pstb	Prefixed Store Byte
pstd	Prefixed Store Doubleword
pstfd	Prefixed Store Floating Double
pstfs	Prefixed Store Floating Single
psth	Prefixed Store Halfword
pstq	Prefixed Store Quadword
pstw	Prefixed Store Word
pstxsd	Prefixed Store VSX Scalar Doubleword
pstxssp	Prefixed Store VSX Scalar Single-Precision
pstxv	Prefixed Store VSX Vector
pstxvp	Prefixed Store VSX Vector Paired
setbc	Set Boolean Condition
setbcr	Set Boolean Condition Reverse
setnbc	Set Negative Boolean Condition
setnbcr	Set Negative Boolean Condition Reverse
stxvp	Store VSX Vector Paired
stxvpx	Store VSX Vector Paired Indexed
stxvrbx	Store VSX Vector Rightmost Byte Indexed
stxvrdx	Store VSX Vector Rightmost Doubleword Indexed
stxvrhx	Store VSX Vector Rightmost Halfword Indexed
stxvrwx	Store VSX Vector Rightmost Word Indexed
vcfuged	Vector Centrifuge Doubleword
vclrlb	Vector Clear Leftmost Bytes
vclrrb	Vector Clear Rightmost Bytes

Table 2.2: Instructions controlled by the v3.0 bit (continued)

vclzdm	Vector Count Leading Zeros Doubleword under bit Mask
vcmpequq[.]	Vector Compare Equal Quadword
vcmpgtsq[.]	Vector Compare Greater Than Signed Quadword
vcmpgtuq[.]	Vector Compare Greater Than Unsigned Quadword
vcmpsq	Vector Compare Signed Quadword
vcmpuq	Vector Compare Unsigned Quadword
vcntmbb	Vector Count Mask Bits Byte
vcntmbd	Vector Count Mask Bits Doubleword
vcntmbh	Vector Count Mask Bits Halfword
vcntmbw	Vector Count Mask Bits Word
vctzdm	Vector Count Trailing Zeros Doubleword under bit Mask
vdivesd	Vector Divide Extended Signed Doubleword
vdivesq	Vector Divide Extended Signed Quadword
vdivesw	Vector Divide Extended Signed Word
vdiveud	Vector Divide Extended Unsigned Doubleword
vdiveuq	Vector Divide Extended Unsigned Quadword
vdiveuw	Vector Divide Extended Unsigned Word
vdivsd	Vector Divide Signed Doubleword
vdivsq	Vector Divide Signed Quadword
vdivsw	Vector Divide Signed Word
vdivud	Vector Divide Unsigned Doubleword
vdivuq	Vector Divide Unsigned Quadword
vdivuw	Vector Divide Unsigned Word
vexpandbm	Vector Expand Byte Mask
vexpanddm	Vector Expand Doubleword Mask
vexpandhm	Vector Expand Halfword Mask
vexpandqm	Vector Expand Quadword Mask
vexpandwm	Vector Expand Word Mask
vextddvlx	Vector Extract Double Doubleword to VSR Left-Indexed
vextddvrx	Vector Extract Double Doubleword to VSR Right-Indexed
vextdubvlx	Vector Extract Double Unsigned Byte to Vector Register Left-Indexed
vextdubvrx	Vector Extract Double Unsigned Byte to Vector Register Right-Indexed
vextduhvlx	Vector Extract Double Unsigned Halfword to Vector Register Left-Indexed
vextduhvrx	Vector Extract Double Unsigned Halfword to Vector Register Right-Indexed
vextduwvlx	Vector Extract Double Unsigned Word to Vector Register Left-Indexed
vextduwvrx	Vector Extract Double Unsigned Word to Vector Register Right-Indexed
vextractbm	Vector Extract Byte Mask
vextractdm	Vector Extract Doubleword Mask
vextracthm	Vector Extract Halfword Mask
vextractqm	Vector Extract Quadword Mask
vextractwm	Vector Extract Word Mask
vextsd2q	Vector Extend Sign Doubleword to Quadword
vgnb	Vector Gather every Nth Bit
vinsblx	Vector Insert Byte from GPR Left-Indexed
vinsbrx	Vector Insert Byte from GPR Right-Indexed
vinsbvlx	Vector Insert Byte from VSR Left-Indexed
vinsbvrx	Vector Insert Byte from VSR Right-Indexed
vinsd	Vector Insert Doubleword from GPR
vinsdlx	Vector Insert Doubleword from GPR Left-Indexed
vinsdrx	Vector Insert Doubleword from GPR Right-Indexed
vinshlx	Vector Insert Halfword from GPR Left-Indexed
vinshrx	Vector Insert Halfword from GPR Right-Indexed
vinshvlx	Vector Insert Halfword from VSR Left-Indexed
vinshvrx	Vector Insert Halfword from VSR Right-Indexed
vinsw	Vector Insert Word from GPR
vinswlx	Vector Insert Word from GPR Left-Indexed
vinswrx	Vector Insert Word from GPR Right-Indexed
vinswvlx	Vector Insert Word from VSR Left-Indexed
vinswvrx	Vector Insert Word from VSR Right-Indexed
vmodsd	Vector Modulo Signed Doubleword
vmodsq	Vector Modulo Signed Quadword
vmodsw	Vector Modulo Signed Word

Table 2.2: Instructions controlled by the v3.0 bit (continued)

vmodud	Vector Modulo Unsigned Doubleword
vmoduq	Vector Modulo Unsigned Quadword
vmoduw	Vector Modulo Unsigned Word
vmsumcud	Vector Multiply-Sum & write Carry-out Unsigned Doubleword
vmulesd	Vector Multiply Even Signed Doubleword
vmuleud	Vector Multiply Even Unsigned Doubleword
vmulhsd	Vector Multiply High Signed Doubleword
vmulhsw	Vector Multiply High Signed Word
vmulhud	Vector Multiply High Unsigned Doubleword
vmulhuw	Vector Multiply High Unsigned Word
vmulld	Vector Multiply Low Doubleword
vmulosd	Vector Multiply Odd Signed Doubleword
vmuloud	Vector Multiply Odd Unsigned Doubleword
vpdepd	Vector Parallel Bits Deposit Doubleword
vpextd	Vector Parallel Bits Extract Doubleword
vrlq	Vector Rotate Left Quadword
vrlqmi	Vector Rotate Left Quadword then Mask Insert
vrlqnm	Vector Rotate Left Quadword then AND with Mask
vsldbi	Vector Shift Left Double by Bit Immediate
vslq	Vector Shift Left Quadword
vsraq	Vector Shift Right Algebraic Quadword
vsrdbi	Vector Shift Right Double by Bit Immediate
vsrq	Vector Shift Right Quadword
vstribl[.]	Vector String Isolate Byte Left-justified
vstribr[.]	Vector String Isolate Byte Right-justified
vstrihl[.]	Vector String Isolate Halfword Left-justified
vstrihr[.]	Vector String Isolate Halfword Right-justified
xscmpeqqp	VSX Scalar Compare Equal Quad-Precision
xscmpgeqp	VSX Scalar Compare Greater Than or Equal Quad-Precision
xscmpgtqp	VSX Scalar Compare Greater Than Quad-Precision
xscvqpsqz	VSX Vector Convert Quad-Precision to Signed Quadword
xscvqpuqz	VSX Vector Convert Quad-Precision to Unsigned Quadword
xscvsqqp	VSX Vector Convert Signed Quadword to Quad-Precision
xscvuqqp	VSX Vector Convert Unsigned Quadword to Quad-Precision
xsmaxcqp	VSX Scalar Maximum Type-C Quad-Precision
xsmincqp	VSX Scalar Minimum Type-C Quad-Precision
xvcvbf16spn	VSX Vector Convert bfloat16 to Single-Precision format Non-signaling
xvcvspbf16	VSX Vector Convert with round Single-Precision to bfloat16 format
xvtlsbb	VSX Vector Test Least-Significant Bit by Byte Operation
xxblendvb	VSX Vector Blend Variable Byte
xxblendvd	VSX Vector Blend Variable Doubleword
xxblendvh	VSX Vector Blend Variable Halfword
xxblendvw	VSX Vector Blend Variable Word
xxeval	VSX Vector Evaluate
xxgenpcvbm	VSX Vector Generate PCV from Byte Mask
xxgenpcvdm	VSX Vector Generate PCV from Doubleword Mask
xxgenpcvhm	VSX Vector Generate PCV from Halfword Mask
xxgenpcvwm	VSX Vector Generate PCV from Word Mask
xxpermx	VSX Vector Permute Extended
xxsplti32dx	VSX Vector Splat Immediate32 Doubleword Indexed
xxspltidp	VSX Vector Splat Immediate Double-Precision
xxspltiw	VSX Vector Splat Immediate Word

Table 2.3: Instructions controlled by the v2.07 bit

addpcis	Add PC Immediate Shifted
bcdcfn.	Decimal Convert From National
bcdcfsq.	Decimal Convert From Signed Quadword
bcdcfz.	Decimal Convert From Zoned
bcdcpsgn.	Decimal CopySign
bcdctn.	Decimal Convert To National
bcdctsq.	Decimal Convert To Signed Quadword
bcdctz.	Decimal Convert To Zoned
bcds.	Decimal Shift
bcdsetsgn.	Decimal Set Sign
bcdsr.	Decimal Shift and Round
bcdtrunc.	Decimal Truncate
bcdus.	Decimal Unsigned Shift
bcdutrunc.	Decimal Unsigned Truncate
cmpeqb	Compare Equal Byte
cmprb	Compare Ranged Byte
cnttzd[.]	Count Trailing Zeros Doubleword
cnttzw[.]	Count Trailing Zeros Word
copy	Copy
cpabort	Copy-Paste Abort
darn	Deliver a Random Number
dtstsfi	DFP Test Significance Immediate
dtstsfiq	DFP Test Significance Immediate Quad
extswsli[.]	Extend Sign Word and Shift Left Immediate
ldat	Load Doubleword Atomic
lwat	Load Word Atomic
lxsd	Load VSX Scalar Doubleword
lxsibzx	Load VSX Scalar as Integer Byte & Zero Indexed
lxsihzx	Load VSX Scalar as Integer Halfword & Zero Indexed
lxssp	Load VSX Scalar Single
lxv	Load VSX Vector
lxvb16x	Load VSX Vector Byte*16 Indexed
lxvh8x	Load VSX Vector Halfword*8 Indexed
lxvl	Load VSX Vector with Length
lxvll	Load VSX Vector Left-justified with Length
lxvwsx	Load VSX Vector Word & Splat Indexed
lxvx	Load VSX Vector Indexed
maddhd	Multiply-Add High Doubleword
maddhdu	Multiply-Add High Doubleword Unsigned
maddld	Multiply-Add Low Doubleword
mcrxrx	Move XER to CR Extended
mffscdrn	Move From FPSCR Control & set DRN
mffscdrni	Move From FPSCR Control & set DRN Immediate
mffsce	Move From FPSCR & Clear Enables
mffscrn	Move From FPSCR Control & set RN
mffscrni	Move From FPSCR Control & set RN Immediate
mffsl	Move From FPSCR Lightweight
mfvsrld	Move From VSR Lower Doubleword
modsd	Modulo Signed Doubleword
modsw	Modulo Signed Word
modud	Modulo Unsigned Doubleword
moduw	Modulo Unsigned Word
mtvsrdd	Move To VSR Double Doubleword
mtvsrws	Move To VSR Word & Splat
paste.	Paste
setb	Set Boolean
stdat	Store Doubleword Atomic
stwat	Store Word Atomic
stxsd	Store VSX Scalar Doubleword
stxsibx	Store VSX Scalar as Integer Byte Indexed
stxsihx	Store VSX Scalar as Integer Halfword Indexed
stxssp	Store VSX Scalar Single

Table 2.3: Instructions controlled by the v2.07 bit (continued)

stxv	Store VSX Vector
stxvb16x	Store VSX Vector Byte*16 Indexed
stxvh8x	Store VSX Vector Halfword*8 Indexed
stxvl	Store VSX Vector with Length
stxvll	Store VSX Vector Left-justified with Length
stxvx	Store VSX Vector Indexed
vabsdub	Vector Absolute Difference Unsigned Byte
vabsduh	Vector Absolute Difference Unsigned Halfword
vabsduw	Vector Absolute Difference Unsigned Word
vbpermd	Vector Bit Permute Doubleword
vclzlsbb	Vector Count Leading Zero Least-Significant Bits Byte
vcmpneb[.]	Vector Compare Not Equal Byte
vcmpneh[.]	Vector Compare Not Equal Halfword
vcmpnew[.]	Vector Compare Not Equal Word
vcmpnezb[.]	Vector Compare Not Equal or Zero Byte
vcmpnezh[.]	Vector Compare Not Equal or Zero Halfword
vcmpnezw[.]	Vector Compare Not Equal or Zero Word
vctzb	Vector Count Trailing Zeros Byte
vctzd	Vector Count Trailing Zeros Doubleword
vctzh	Vector Count Trailing Zeros Halfword
vctzlsbb	Vector Count Trailing Zero Least-Significant Bits Byte
vctzw	Vector Count Trailing Zeros Word
vextractd	Vector Extract Doubleword
vextractub	Vector Extract Unsigned Byte
vextractuh	Vector Extract Unsigned Halfword
vextractuw	Vector Extract Unsigned Word
vextsb2d	Vector Extend Sign Byte To Doubleword
vextsb2w	Vector Extend Sign Byte To Word
vextsh2d	Vector Extend Sign Halfword To Doubleword
vextsh2w	Vector Extend Sign Halfword To Word
vextsw2d	Vector Extend Sign Word To Doubleword
vextublx	Vector Extract Unsigned Byte Left-Indexed
vextubrx	Vector Extract Unsigned Byte Right-Indexed
vextuhlx	Vector Extract Unsigned Halfword Left-Indexed
vextuhrx	Vector Extract Unsigned Halfword Right-Indexed
vextuwlx	Vector Extract Unsigned Word Left-Indexed
vextuwrx	Vector Extract Unsigned Word Right-Indexed
vinsertb	Vector Insert Byte
vinsertd	Vector Insert Doubleword
vinserth	Vector Insert Halfword
vinsertw	Vector Insert Word
vmul10cuq	Vector Multiply-by-10 & write Carry Unsigned Quadword
vmul10ecuq	Vector Multiply-by-10 Extended & write Carry Unsigned Quadword
vmul10euq	Vector Multiply-by-10 Extended Unsigned Quadword
vmul10uq	Vector Multiply-by-10 Unsigned Quadword
vnegd	Vector Negate Doubleword
vnegw	Vector Negate Word
vpermr	Vector Permute Right-indexed
vprtybd	Vector Parity Byte Doubleword
vprtybq	Vector Parity Byte Quadword
vprtybw	Vector Parity Byte Word
vrldmi	Vector Rotate Left Doubleword then Mask Insert
vrldnm	Vector Rotate Left Doubleword then AND with Mask
vrlwmi	Vector Rotate Left Word then Mask Insert
vrlwnm	Vector Rotate Left Word then AND with Mask
vslv	Vector Shift Left Variable
vsrv	Vector Shift Right Variable
wait	Wait
xsabsqp	VSX Scalar Quad-Precision Absolute
xsaddqp[o]	VSX Scalar Quad-Precision Add [& round to Odd]
xscmpexpdp	VSX Scalar Double-Precision Compare Exponents
xscmpexpqp	VSX Scalar Quad-Precision Compare Exponents

Table 2.3: Instructions controlled by the v2.07 bit (continued)

xscmpoqp	VSX Scalar Quad-Precision Compare Ordered
xscmpuqp	VSX Scalar Quad-Precision Compare Unordered
xscpsgnqp	VSX Scalar Quad-Precision CopySign
xscvdpqp	VSX Scalar Quad-Precision Convert From Double-Precision
xscvqpdp[o]	VSX Scalar round & Convert Quad-Precision to Double-Precision [using round to Odd]
xscvqpsdz	VSX Scalar truncate & Convert Quad-Precision to Signed Doubleword
xscvqpswz	VSX Scalar truncate & Convert Quad-Precision to Signed Word
xscvqpudz	VSX Scalar truncate & Convert Quad-Precision to Unsigned Doubleword
xscvqpuwz	VSX Scalar truncate & Convert Quad-Precision to Unsigned Word
xscvsdqp	VSX Scalar Convert Signed Doubleword format to Quad-Precision format
xscvsphp	VSX Scalar round & Convert Double-Precision to Half-Precision
xscvudqp	VSX Scalar Convert Unsigned Doubleword format to Quad-Precision format
xsdivqp[o]	VSX Scalar Quad-Precision Divide [& round to Odd]
xsiexpdp	VSX Scalar Double-Precision Insert Exponent
xsiexpqp	VSX Scalar Quad-Precision Insert Exponent
xsmaddqp[o]	VSX Scalar Quad-Precision Multiply-Add [& round to Odd]
xsmsubqp[o]	VSX Scalar Quad-Precision Multiply-Subtract [& round to Odd]
xsmulqp[o]	VSX Scalar Quad-Precision Multiply [& round to Odd]
xsnabsqp	VSX Scalar Quad-Precision Negative Absolute
xsnegqp	VSX Scalar Quad-Precision Negate
xsnmaddqp[o]	VSX Scalar Quad-Precision Negative Multiply-Add [& round to Odd]
xsnmsubqp[o]	VSX Scalar Quad-Precision Negative Multiply-Subtract [& round to Odd]
xsrqpi	VSX Scalar Round to Quad-Precision Integer
xsrqpix	VSX Scalar Round to Quad-Precision Integer with Inexact
xsrqpxp	VSX Scalar Quad-Precision Round to Double-Extended-Precision
xssqrtqp[o]	VSX Scalar Quad-Precision Square Root [& round to Odd]
xssubqp[o]	VSX Scalar Quad-Precision Subtract [& round to Odd]
xststdcdp	VSX Scalar Double-Precision Test Data Class
xststdcqp	VSX Scalar Quad-Precision Test Data Class
xststdcsp	VSX Scalar Single-Precision Test Data Class
xsxexpdp	VSX Scalar Double-Precision Extract Exponent
xsxexpqp	VSX Scalar Quad-Precision Extract Exponent
xsxsigdp	VSX Scalar Double-Precision Extract Significand
xsxsigqp	VSX Scalar Quad-Precision Extract Significand
xvcvhpsp	VSX Vector Convert Half-Precision to Single-Precision
xvcvsphp	VSX Vector round & Convert Single-Precision to Half-Precision
xviexpdp	VSX Vector Double-Precision Insert Exponent
xviexpsp	VSX Vector Single-Precision Insert Exponent
xvtstdcdp	VSX Vector Double-Precision Test Data Class
xvtstdcsp	VSX Vector Single-Precision Test Data Class
xvxexpdp	VSX Vector Double-Precision Extract Exponent
xvxexpsp	VSX Vector Single-Precision Extract Exponent
xvxsigdp	VSX Vector Double-Precision Extract Significand
xvxsigsp	VSX Vector Single-Precision Extract Significand
xxbrd	VSX Vector Byte-Reverse Doubleword
xxbrh	VSX Vector Byte-Reverse Halfword
xxbrq	VSX Vector Byte-Reverse Quadword
xxbrw	VSX Vector Byte-Reverse Word
xxextractuw	VSX Vector Extract Unsigned Word
xxinsertw	VSX Vector Insert Word
xxperm	VSX Vector Permute
xxpermr	VSX Vector Permute Right-indexed
xxspltib	VSX Vector Splat Immediate Byte

2.6 Other Hypervisor Resources

In addition to the resources described in the preceding sections, all hypervisor privileged instructions as well as the following resources are hypervisor resources, accessible to software only when the thread is in hypervisor state except as noted below.

- All implementation-specific resources except for privileged non-hypervisor implementation-specific SPRs. (See Section 5.4.4 for the list of the implementation-specific SPRs that are allowed to be privileged non-hypervisor SPRs.) Implementation-specific registers include registers (e.g., "HID" registers) that control hardware functions or affect the results of instruction execution. Examples include resources that disable caches, disable hardware error detection, set breakpoints, control power management, or significantly affect performance.
- ME bit of the MSR
- SPRs defined as hypervisor-privileged in Section 5.4.4. (Note: Although the Time Base, the PURR, and the SPURR can be altered only by a hypervisor program, the Time Base can be read by all programs and the PURR and SPURR can be read when the thread is in privileged state.)

The contents of a hypervisor resource can be modified by the execution of an instruction (e.g., ***mtspr***) only in hypervisor state ($MSR_{HV\ PR}$ = 0b10). An attempt to modify the contents of a given hypervisor resource, other than MSR_{ME}, in privileged but non-hypervisor state ($MSR_{HV\ PR}$ = 0b00) causes a Privileged Instruction type Program Interrupt when $LPCR_{EVIRT}$=0 and a Hypervisor Emulation Assistance interrupt when $LPCR_{EVIRT}$=1. An attempt to modify MSR_{ME} in privileged but non-hypervisor state is ignored (i.e., the bit is not changed).

Programming Note

Because the SPRs listed above are privileged for writing, an attempt to modify the contents of any of these SPRs in problem state (MSR_{PR}=1) using ***mtspr*** causes a Privileged Instruction type Program exception, and similarly for MSR_{ME}.

Architecture Note

Any resource having the property that alteration of the resource by a thread in one partition could affect the integrity of other partitions must be a hypervisor resource.

In general, control bits that are hypervisor resources should not be defined in registers or resources that contain bits that can be altered by non-hypervisor programs.

2.7 Sharing Hypervisor and Ultravisor Resources

Shared SPRs are SPRs that are accessible to multiple threads. Changes to shared SPRs made by one thread are immediately readable (using ***mfspr***) by all other threads sharing the SPR.

The LPIDR and DPDES must appear to software to be shared among threads of a sub-processor (see Section 2.8). If the implementation does not support sub-processors, the LPIDR and DPDES must be shared among all threads of the multi-threaded processor.

Certain additional hypervisor and ultravisor resources, and the PVR, may be shared among threads. Programs that modify these resources must be aware of this sharing, and must allow for the fact that changes to these resources may affect more than one thread.

The following additional resources may be shared among threads.

- HRMOR (see Section 2.3)
- LPIDR (see Section 2.4)
- PCR (see Section 2.5)
- URMOR (see Section 3.2)
- PVR (see Section 5.3.1)
- RPR (see Section 5.3.7)
- PTCR (see Section 6.7.6.1)
- AMOR (see Section 6.7.13.1)
- HMEER (see Section 7.2.11)
- Time Base (see Section 8.2)
- Virtual Time Base (see Section 8.3)
- Hypervisor Decrementer (see Section 8.5)
- certain implementation-specific registers or implementation-specific fields in architected registers

The set of resources that are shared is implementation-dependent.

Threads that share any of the resources listed above, with the exception of the PTCR, the PVR, the URMOR, and the HRMOR, must be in the same partition.

For each field of the LPCR, except the AIL, EVIRT, ONL, HDICE, MER,PECE, HEIC, and HVICE fields, software must ensure that the contents of the field are identical among all threads that are in the same partition and are not in hypervisor state.

Software must ensure that the contents of UILE and $SMFCTRL_E$ are identical among all threads in the system that have completed ultravisor initialization. The contents of the D and UDEE fields of SMFCTRL may differ among threads.

2.8 Sub-Processors

Hardware is allowed to sub-divide a multi-threaded processor into “sub-processors” that appear to privileged programs as multi-threaded processors with fewer threads. Such a multi-threaded processor appears to the hypervisor as a processor with a number of threads equal to the sum of all sub-processor threads, and in which the LPIDR for each sub-processor must appear to be shared among all threads of that sub-processor.

2.9 Thread Identification Register (TIR)

The TIR is a 64-bit read-only register that contains the thread number, which is a binary number corresponding to the thread.

For implementations that do not support sub-processors, the thread number of a thread is unique among all thread numbers of threads on the multi-threaded processor.

For implementations that support sub-processors, the value of this register depends on whether it is read in hypervisor or privileged, non-hypervisor state as follows.

- When this register is read in privileged, non-hypervisor state, the thread number is unique among all thread numbers of threads on the sub-processor.
- When this register is read in hypervisor state, the thread number is unique among all thread numbers of threads on the multi-threaded processor.

Threads are numbered sequentially, with valid values ranging from 0 to t-1, where t is the number of threads implemented. A thread for which TIR = n is referred to as “thread n.”

The layout of the TIR is shown below.

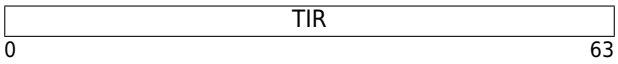


Figure 2.4: Thread Identification Register

Access to the TIR is privileged.

Since the thread number contained in this register is different if it is read in hypervisor state from when it is read in privileged, non-hypervisor state in implementations that support sub-processors, the following conventions are used.

- The value returned in privileged, non-hypervisor state is referred to as the “privileged thread number.”
- The value returned in hypervisor state is referred to as the “hypervisor thread number.”

2.10 Hypervisor Interrupt Little-Endian (HILE) Bit

The Hypervisor Interrupt Little-Endian (HILE) bit is a bit in an implementation-dependent register or similar mechanism. The contents of the HILE bit are copied into MSR_{LE} by interrupts that result in $MSR_{S\ HV}$ being equal to 0b01 (see Section 7.5), to establish the Endian mode for the interrupt handler. The HILE bit is set, by an implementation-dependent method, only during system initialization.

The contents of the HILE bit must be the same for all threads under the control of a given instance of the hypervisor; otherwise all results are undefined.

Chapter 3. Ultravisor and Secure Memory Facility (SMF)

3.1 Overview

The Secure Memory Facility (SMF) provides secure isolation of partitions from one another and from higher privilege system software. SMF functionality is implemented using a combination of hardware facilities and firmware that runs at a privilege level above the hypervisor. SMF targets a threat model in which the hypervisor can be compromised such that its inherent isolation capabilities can no longer be counted on. Maintaining the security of data is the sole objective of the ultravisor. It has no role in platform management and is not expected to deal with denial of service attacks. References elsewhere in the Books to "secure systems" apply more generally, and do not necessarily imply that the system uses SMF.

The SMF protection mechanism is based on the assignment of partitions to security domains. The hypervisor is in one security domain, along with all processes that run directly under the hypervisor and all partitions that do not take advantage of the SMF security capabilities. Each of the secure partitions is assigned to its own security domain so that its data and instructions can be protected from access by other security domains. A partition is identified as secure when MSR_S=1. Each location in main storage has an associated Secure Memory property, mem_{SM}. Memory with mem_{SM}=1 may be referred to as "secure memory." Memory with mem_{SM}=0 may be referred to as "ordinary memory." The granularity and method with which main storage is mapped for the Secure Memory property is implementation specific. The Secure Memory property is commonly cached in the TLB and in implementation-specific lookaside buffers. When secure data are to be shared with untrusted software, the standard synchronization associated with PTE updates is used to regulate access. For example, prior to sharing secure data, the PTEs used to access the data are marked invalid and the corresponding TLB entires invalidated by the ultravisor using the standard invalidation sequence. (See Section 6.10.1.2.) The data are then encrypted and made available in ordinary memory (either mem_{SM} is turned off or the data are moved to ordinary memory). Finally the PTEs that will be used to access the data in ordinary memory are marked valid. (The last step may be done lazily.) Software running with MSR_S=0 is prohibited from accessing secure memory. Software running with MSR_S=1 may access both secure and ordinary memory.

Programming Note

The ultravisor will commonly use a no-execute protection setting to prevent a secure partition from executing instructions from any ordinary memory mapped into its address space.

Engineering Note

The preferred method to implement the mem_{SM} checking in the TLB and ERAT is to include MSR_S in the entry, only creating the entry when mem_{SM} permits the access. If the bit is included in the tag portion of the array, the array can contain entries for both secure and ordinary software concurrently. If the bit is included in the RAM portion, the array can contain either a secure or an ordinary entry, but not both. Since secure software can access both secure and ordinary memory, it is not required to evaluate mem_{SM} to create a secure entry.

SMF firmware runs in ultravisor state, a privilege level above that of the hypervisor. That firmware, along with the SMF hardware, is responsible for maintaining isolation of secure partitions from each other and from the hypervisor. This is accomplished by direct ultravisor management of the partition-scoped translation tables in secure memory for secure partitions. The ultravisor itself runs only in (ultravisor) real addressing mode. Security is the result of proper management of the partition-scoped translation together with the hardware enforcement of the access restriction for secure memory. With this hybrid approach, firmware has the ability to enable secure memory sharing between secure partitions and ordinary memory sharing between a given secure partition and the hypervisor, e.g. for system calls. The ultravisor can access any architecture resource or facility.

The hypervisor is expected to cooperate in the management of secure partitions by using ultravisor calls to dispatch them and to manage their storage allocations. To protect against programming errors and malicious hypervisor behavior, ***mtmsr***[***d***], ***rfid***, ***hrfid***, and ***rfscv*** preserve MSR_S and hypervisor interrupts from secure partitions are always received in ultravisor state.

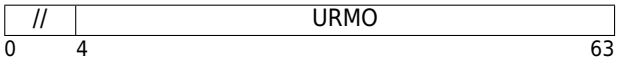
The purpose of intercepting hypervisor interrupts is to protect the state of the secure partition from the hypervisor. The ultravisor's interrupt handler provides a 'shim' that saves and clears the processing state, and then transfers control to the hypervisor to handle the exception condition itself. The ultravisor will restore the secure partition state when it services the ultravisor call to (re-) dispatch the secure partition. Note that the ultravisor's goal is merely to protect the security of data, and not to provide broader system management oversight.

Engineering Note

Care must be taken in the classification of resources used for platform control and management as ultravisor or hypervisor resources. Platform control and management are generally functions of the hypervisor. However, BARs (Base Address Registers) may have the ability to establish the security of their associated facilities. Other such cases may exist.

3.2 Ultravisor Real Mode Offset Register (URMOR)

The layout of the Ultravisor Real Mode Offset Register (URMOR) is shown in Figure 3.1 below.



Bit(s)	Name	Description
4:63	URMO	Real Mode Offset

Figure 3.1: Ultravisor Real Mode Offset Register

All other fields are reserved.

The supported URMO values are the non-negative multiples of 2^r, where r is the same implementation-dependent value that constrains the HRMO field of the HRMOR.

The contents of the URMOR affect how some storage accesses are performed as described in Sections 6.7.3 and 6.7.5.

3.3 Ultravisor Interrupt Little-Endian (UILE) Bit

The Ultravisor Interrupt Little-Endian (UILE) bit is a bit in an implementation-dependent register or similar mechanism. The contents of the UILE bit are copied into MSR_{LE} by interrupts that result in $MSR_{S\,HV}$ being equal to 0b11 (see Section 7.5), to establish the Endian mode for the interrupt handler. The UILE bit is set, by an implementation-dependent method, only during system initialization.

The contents of the UILE bit must be the same for all threads in the system; otherwise all results are undefined.

3.4 Secure Memory Facility Control Register (SMFCTRL)

The Secure Memory Facility Control Register (SMFCTRL) is shown in Figure 3.2 below.

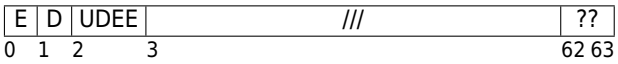


Figure 3.2: Secure Memory Facility Control Register (SMFCTRL)

Bit(s) Definition

0 **SMF Enable** (E)
- 0 SMF functionality including secure memory checking is disabled.
- 1 SMF functionality including secure memory checking is enabled.

When $SMFCTRL_E$=1, writing the PTCR is ultravisor privileged.

1 **Debug enable** (D)
- 0 Ultravisor debug mode is disabled.
- 1 Ultravisor debug mode is enabled.

In ultravisor debug mode, CIABR, DAWRn, and DAWRXn are ultravisor privileged. See Chapter 10 for a description of how instruction and data address tracing work in ultravisor debug mode.

2 **Ultravisor Doorbell Exit Enable** (UDEE)
- 0 When the ***stop*** instruction is executed with $PSSCR_{EC}$=1, Directed Ultravisor Doorbell exceptions are not enabled to cause exit from power-saving mode
- 1 When the ***stop*** instruction is executed with $PSSCR_{EC}$=1, Directed Ultravisor Doorbell exceptions are enabled to cause exit from power-saving mode.

3:61 Reserved

62:63 **Implementation-specific use**

$SMFCTRL_E$ must be set to 1 prior to exiting ultravisor state if the system will use the SMF facilities. (When $SMFCTRL_E$=0 and MSR_S=0, there is no way to achieve MSR_S=1 without a reboot.)

Programming Note

The two useful runtime states with respect to SMF operation are (1) MSR_S=0 and $SMFCTRL_E$=0 (SMF permanently disabled) and (2) $SMFCTRL_E$=1 (SMF enabled). Very limited verification may be performed on the state with MSR_S=1 and $SMFCTRL_E$=0 and around state changes of $SMFCTRL_E$. Therefore, software should change the value of $SMFCTRL_E$ at most once, making the change prior to the first dispatch of a partition, and spending as little time as possible in the state with MSR_S=1 and $SMFCTRL_E$=0.

If $SMFCTRL_E$=0, $SMFCTRL_D$ and $SMFCTRL_{UDEE}$ must be set to zero. References to $SMFCTRL_D$=1 or $SMFCTRL_{UDEE}$=1 elsewhere in the architecture assume $SMFCTRL_E$=1 unless otherwise stated or obvious from context.

3.4.1 Enabling SMF and Secure Memory Enforcement

The $SMFCTRL_E$ bit enables SMF functionality. When $SMFCTRL_E$=1, certain facilities are ultravisor resources instead of hypervisor resources and secure memory checking is enabled.

Independent of the basic feature enablement above, SMF has state transition rules that facilitate the protection of security domains. (While these rules are nominally independent of the value of $SMFCTRL_E$, some transitions cannot happen when $SMFCTRL_E$=0. Specifically, ultravisor interrupts cannot occur when $SMFCTRL_E$=0.)

- All interrupts that are not ultravisor interrupts preserve MSR_S. (Ultravisor interrupts necessarily set MSR_S to 1.)
- ***mtmsr***[***d***], ***rfid***, ***hrfid***, and ***rfscv*** are not permitted to change MSR_S.

Table 3.1 summarizes the effect of the $SMFCTRL_E$ bit and the $MSR_{S\ HV\ PR}$ bits on various facilities.

facility	$MSR_{S\ HV\ PR}$	$SMFCTRL_E$	$LPCR_{EVIRT}$	behavior
mtspr or ***mfspr*** specifying URMOR, USRR0, USRR1, USPRG0, USPRG1, or SMFCTRL; ***urfid***, ***msgsndu***, ***msgclru***	110	dc	dc	execution allowed
	all xxx except 110 **	dc	dc	Privileged Instruction type Program interrupt to xx0
mtspr specifying PTCR	110	dc	dc	execution allowed
	010	0	dc	execution allowed
		1	dc	HEAI to 010
	x00	dc	0	Privileged Instruction type Program interrupt to x00
			1	HEAI to x10
	xx1**	dc	dc	Privileged Instruction type Program interrupt to xx0
mtspr or ***mfspr*** specifying DAWRn, DAWRXn or CIABR when $SMFCTRL_D$=1	110	1	dc	execution allowed
	010	1	dc	HEAI to 010
	x00	1	0	Privileged Instruction type Program interrupt to x00
			1	HEAI to x10
	xx1**	1	dc	Privileged Instruction type Program interrupt to xx0
sc 2 instruction	dc**	0	dc	hypervisor call, but with SRR1 showing LEV=2
	dc**	1	dc	ultravisor call
mem_{SM} evaluation and match	dc**	0	dc	disabled
	dc**	1	dc	enabled*
* mem_{SM} evaluation may be avoided when MSR_S=1, depending on translation cache design dc = don't care ** The encoding $MSR_{S\ HV\ PR}$=0b111 is reserved and must not be used.				

Table 3.1: Ultravisor Resource Behavior

Programming Note

Access to memory by mechanisms outside the core must also enforce secure memory access restrictions. Facilities that translate addresses or otherwise use real addresses to access memory must check mem_{SM} against $PATE_S$ for the partition on behalf of which they access memory.

Such mechanisms will require a means to evaluate mem_{SM} and a proxy for $SMFCTRL_E$ to provide the same enablement function for secure memory access enforcement as in the core.

In addition or as an alternative, TCE tables may be managed by the ultravisor and used to identify regions of memory that I/O devices may access.

Chapter 4. Branch Facility

4.1 Branch Facility Overview

This chapter describes the details concerning the registers and the privileged instructions implemented in the Branch Facility that are not covered in Book I.

4.2 Branch Facility Registers

4.2.1 Machine State Register

The Machine State Register (MSR) is a 64-bit register. This register defines the state of the thread. On interrupt, the MSR bits are altered in accordance with Figure 7.15 on page 1202. The MSR can also be modified by the ***mtmsr***[***d***], ***rfscv***, ***rfid***, ***hrfid***, and ***urfid*** instructions. It can be read by the ***mfmsr*** instruction.

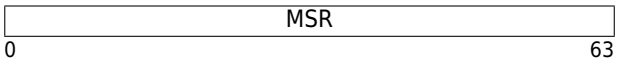


Figure 4.1: Machine State Register

Below are shown the bit definitions for the Machine State Register.

Architecture Note

Defined MSR bits are classified as either full function or partial function. Full function MSR bits are saved in SRR1, HSRR1, or USRR1 when an interrupt other than a System Call Vectored interrupt occurs and restored by ***rfscv***, ***rfid***, ***hrfid***, or ***urfid*** while partial function MSR bits are not saved or restored. Full function MSR bits lie in the range 0:32, 37:41, and 48:63, and partial function MSR bits lie in the range 33:36 and 42:47.

Reserved MSR bits in the "full function range" need not be saved or restored. If they are not restored then they must be written as 0; if they are not saved then for System Call Vectored interrupt the corresponding CTR bits must be written as 0, and for interrupts other than System Call Vectored interrupt the corresponding SRR1, HSRR1, or USRR1 bits must be written as 0. The properties of reserved bits in System Registers are such that this alternative behavior does not conflict with the descriptions of ***sc***, ***scv***, ***rfscv***, ***rfid***, ***hrfid***, ***urfid***, and interrupt processing elsewhere in this Book.

Bit(s)	Definition
0	**Sixty-Four-Bit Mode** (SF) 0 The thread is in 32-bit mode. 1 The thread is in 64-bit mode. Software must ensure that SF=1 whenever the thread is in ultravisor state.
1:2	Reserved

Architecture Note

Bit 2 will be among the last to be assigned a meaning. It was the ISF (Interrupt Sixty-Four Bit Mode) bit in earlier versions of the architecture.

3	**Hypervisor State** (HV) 0 The thread is not in hypervisor state. 1 If MSR_{PR}=0, the thread is in hypervisor state; otherwise the thread is not in hypervisor state.

Programming Note

The privilege state of the thread is determined by MSR_S, MSR_{HV}, and MSR_{PR}, as follows.

S	HV	PR	
0	x	1	problem
1	0	1	problem
x	x	0	privileged
x	1	0	hypervisor
1	1	0	ultravisor
1	1	1	reserved

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. Ultravisor state is also a hypervisor state ($MSR_{HV\ PR}$ = 0b10). All references to "hypervisor state" in the Books include ultravisor 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***, ***hrfid***, and ***urfid***.

It is possible to run an operating system in an environment that lacks a hypervisor, by always having MSR_{HV} = 1 and using $MSR_{HV\ PR}$ = 0b10 for the operating system (effectively, the OS runs in hypervisor state) and $MSR_{HV\ PR}$ = 0b11 for applications. In this use, MSR_S would be 0, and the environment would also lack an ultravisor.

4 Reserved

5 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***, ***urfid***, and ***rfscv*** instructions is such that, unless software deliberately sets the bit to 1, the bit will continue to contain 0.

Architecture Note

Bit 5 was the Split Little Endian (SLE) bit in V. 2.07 of the architecture. The 0 value of the bit caused the Endian mode of the thread to be the mode specified by MSR_{LE}, for both instruction fetch and data access, which is the behavior required by preceding and subsequent versions of the architecture. The SLE function was removed from the architecture in V. 2.07B, in a manner consistent with the fact that POWER8 processors implement the function. This bit will not be assigned a meaning in versions of the architecture subsequent to V. 2.07 except after careful consideration of the effect of such assignment on POWER8.

6:37 Reserved

Architecture Note

Bits 29:31 will be among the last to be assigned a meaning. In earlier versions of the architecture bits 29:30 were the TS (Transaction State) field and bit 31 was the TM (Transactional Memory Available) bit.

Architecture Note

In the POWER Architecture, $SRR1_5$ is used by the Instruction Storage interrupt to indicate a loop in the translation mechanism. Because of this, MSR_{37} will not be assigned a new meaning in the near future.

38 **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 **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.

Programming Note

An application binary interface defined to support Vector-Scalar operations should also specify a requirement that MSR_{FP} and MSR_{VEC} be set to 1 whenever MSR_{VSX} is set to 1.

41 **Secure** (S)

0 The thread is not in Secure state. It may not access Secure memory. The thread is not in ultravisor state.

1 The thread is in Secure state. If MSR_{HV}=1 and MSR_{PR}=0, the thread is in ultravisor state; otherwise the value does not affect privilege. The state with MSR_{HV}=1 and MSR_{PR}=1 is reserved. Software must not set $MSR_{S\ HV\ PR}$ = 0b111. References elsewhere in this document to $MSR_{HV\ PR}$=0b11 assume MSR_S=0 unless otherwise stated or obvious from context.

Programming Note

MSR_S can be set to 1 only by the *System Call* instruction and some interrupts. It can be set to 0 only by ***urfid***.

Any instruction or event that causes $MSR_{S\ HV\ PR}$ to be set to 0b110 also causes MSR_{IR} and MSR_{DR} to be set to 0.

Engineering Note

The primary reason the state with $MSR_{S\ HV\ PR}$=0b111 is reserved is for verification and test savings. A secondary reason is that it isn't clear whether the most useful definition would be a secure process running under the hypervisor (would require the addition of fine grained security to the architecture) or a process running under the ultravisor.

If secure platform initialization would be aided by running in problem state under the ultravisor, platform developers may choose to offer that capability, strictly limited to the platform initialization process, with the understanding that the architecture might be extended in a manner incompatible with that use at some time in the future.

42:47 Reserved

Architecture Note

Bit 45 will be among the last to be assigned a meaning. In earlier versions of the architecture bit 45 was the POW (Power Management) bit.

Bit 46 is used on some implementations for an implementation-specific function. (Bit 46 is the Temporary GPR Remapping (TGPR) bit on the 603.)

48 **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, Directed Hypervisor Doorbell, and Directed Ultravisor Doorbell interrupts are enabled; see Section 7.5.12 on page 1217, Section 7.5.19 on page 1228, Section 7.5.20 on page 1229, and Section 7.5.28 on page 1232.

49 **Problem State** (PR)

0 The thread is in privileged state.

1 If $MSR_{S\ HV} \neq$ 0b11, the thread is in problem state.

Programming Note

Any instruction that sets MSR_{PR} to 1 also sets MSR_{EE}, MSR_{IR}, and MSR_{DR} to 1.

The state with $MSR_{S\ HV\ PR}$=0b111 is reserved.

50 **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.

51 **Machine Check Interrupt Enable** (ME)

0 Machine Check interrupts are disabled.

1 Machine Check interrupts are enabled.

This bit is a hypervisor resource; see 2, "Logical Partitioning (LPAR) and Thread Control", on page 1054.

Programming Note

The only instructions that can alter MSR_{ME} are ***rfid***, ***hrfid***, and ***urfid***.

52 ***Floating-Point Exception Mode 0*** (FE0)

See below.

53:54 **Trace Enable** (TE)

00 Trace Disabled: The thread executes instructions normally.

01 Branch Trace: The thread generates a Branch type Trace interrupt after completing the execution of a branch instruction, whether or not the branch is taken.

10 Single Step Trace: The thread generates a Single-Step type Trace interrupt after successfully completing the execution of the next instruction, unless that instruction is a ***urfid***, ***hrfid***, ***rfid***, ***rfscv***, or a *Power-Saving Mode* instruction, all of which are never traced. Successful completion means that the instruction caused no other interrupt.

11 Reserved.

Branch tracing need not be supported. If the function is not implemented, the 0b01 bit encoding is treated as reserved.

55 **Floating-Point Exception Mode 1** (FE1)

See below.

56:57 Reserved

Architecture Note

Bit 57 will be among the last to be assigned a meaning. It was the IP (Interrupt Prefix) bit in earlier versions of the architecture.

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 MSR_S and in the definition of MSR_{PR}.

Programming Note

Before hypervisor support was added to the architecture, "translation is disabled" for MSR_{IR}=0 truly meant that no translation was performed for instruction addresses, and correspondingly for MSR_{DR}=0 for data addresses. The architecture continues to use "translation is disabled" to refer to MSR_{IR}=0 and MSR_{DR}=0 despite that the behavior today is more complicated. When $MSR_{HV\ IR}$=0b10, it is still true that no translation is performed for instruction addresses, and correspondingly for data addresses if $MSR_{HV\ DR}$=0b10. But in privileged non-hypervisor state when MSR_{IR}=0 or MSR_{DR}=0, limited translation is performed under control of the hypervisor. For an HPT translation guest, translation is performed as described in Section 6.7.3.3, with storage exceptions directed to the hypervisor. For a Radix Tree Translation guest, only partition-scoped translation is performed, with storage exceptions directed to the hypervisor.

59 **Data Relocate** (DR)

0 Data address translation is disabled.
1 Data address translation is enabled.

Programming Note

See the second Programming Note in the definition of MSR_{IR} and the Programming Notes in the definition of MSR_S and in the definition of MSR_{PR}.

60 Reserved

61 **Performance Monitor Mark** (PMM)

This bit is used by software in conjunction with the Performance Monitor, as described in Chapter 11.

Programming Note

Software can use this bit as a process-specific marker which, in conjunction with $MMCR0_{FCM0\ FCM1}$ (see Section 11.4.4) and MMCR2 (see Section 11.4.6), permits events to be counted on a process-specific basis. (The bit is saved by interrupts and restored by ***rfid***.)

Common uses of the PMM bit include the following.

- All counters count events for a few selected processes. This use requires the following bit settings.
  - MSR_{PMM}=1 for the selected processes, MSR_{PMM}=0 for all other processes
  - $MMCR0_{FCM0}$=1
  - $MMCR0_{FCM1}$=0
  - MMCR2 = 0x0000
- All counters count events for all but a few selected processes. This use requires the following bit settings.
  - MSR_{PMM}=1 for the selected processes, MSR_{PMM}=0 for all other processes
  - $MMCR0_{FCM0}$=0
  - $MMCR0_{FCM1}$=1
  - MMCR2 = 0x0000

Notice that for both of these uses a mark value of 1 identifies the "few" processes and a mark value of 0 identifies the remaining "many" processes. Because the PMM bit is set to 0 when an interrupt occurs (see Figure 7.15 on page 1202), interrupt handlers are treated as one of the "many". If it is desired to treat interrupt handlers as one of the "few", the mark value convention just described would be reversed.

If only a specific counter n is to be frozen, $MMCR0_{FCM0\ \ FCM1}$ is set to 0b00, and $MMCR2_{FCnM0}$ and $MMCR2_{FCnM1}$ instead of $MMCR0_{FCM0}$ and $MMCR0_{FCM1}$ are set to the values described above.

Engineering Note

The two mark values (0 and 1) are equivalent except with respect to interrupts. That is, either mark value can be specified for a given process, and either mark value can control whether the PMCs are incremented, but interrupts always cause the mark value in the MSR to be set to 0 (see Figure 7.15 on page 1202).

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 7.4.3, "Interrupt Processing"

on page 1199, 7.5.1, “System Reset Interrupt” on page 1204, and 7.5.2, “Machine Check Interrupt” on page 1206.

63 **Little-Endian Mode** (LE)

0 The thread is in Big-Endian mode.
1 The thread is in Little-Endian mode.

Programming Note

The only instructions that can alter MSR_{LE} are ***rfid***, ***hrfid***, ***urfid***, and ***rfscv***.

The Floating-Point Exception Mode bits FE0 and FE1 are interpreted as shown below. For further details see Book I.

FE0	FE1	Mode
0	0	Ignore Exceptions
0	1	Imprecise Nonrecoverable
1	0	Imprecise Recoverable
1	1	Precise

The initial state of the MSR should be as follows:

Bit(s)	Name	Value
0	SF	1
1		0
2		unspecified
3	HV	1
4		unspecified
5	0	0
6:28		unspecified
29:31		0b000
32:37		unspecified
38	VEC	0
39		unspecified
40	VSX	0
41	S	1
42:47		unspecified
48	EE	0
49	PR	0
50	FP	0
51	ME	0
52	FE0	0
53:54	TE	0b00
55	FE1	0
56:57		unspecified
58	IR	0
59	DR	0
60		unspecified
61	PMM	unspecified
62	RI	0
63	LE	0

4.2.2 Processor Stop Status and Control Register (PSSCR)

The layout of the PSSCR is shown below.

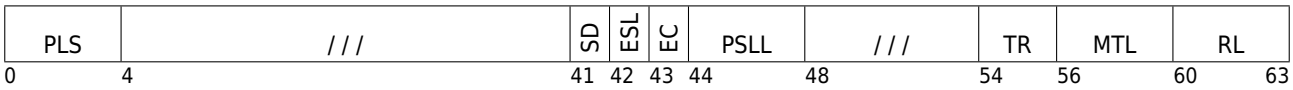


Figure 4.2: Processor stop Status and Control Register

The contents of the PSSCR control the operation of the ***stop*** instruction and provide status indicating the level of power saving that was entered while in power-saving mode.

All fields of this register can be read and written by the hypervisor using either hypervisor SPR 855 or privileged SPR 823. A subset of the fields of this register can be read and written in privileged non-hypervisor state using privileged SPR 823, as specified below. Fields that can only be read or written by the hypervisor are indicated below; all other fields can be read or written in either privileged non-hypervisor or hypervisor states. When a field that is accessible only to the hypervisor is accessed in privileged non-hypervisor state, writes have no effect and reads return 0s regardless of the value of the field.

The bits and their meanings are as follows.

Bit(s) Definition

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 systems 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 7.5.26.

For power-saving levels that allow loss of the LPCR, implementations must provide the means to exit power-saving mode upon the occurrence of a System Reset exception and any of the exceptions that were enabled by the PECE field when the ***stop*** instruction was executed. For this case, the implementation is also allowed to exit on the occurrence of any exceptions that were disabled by the PECE as well.

For power-saving levels that allow loss of SMFCTRL, implementations must provide the means to exit power-saving mode upon the occurrence of a Directed Ultravisor Doorbell exception if $SMFCTRL_{UDEE}$ was set to 1 when ***stop*** was executed. For this case, the implementation is also allowed to exit on the occurrence of a Directed Ultravisor Doorbell exception if $SMFCTRL_{UDEE}$ was set to 0 when ***stop*** was executed.

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 4.3.2.

Engineering Note

When the thread is at a power-saving level that permits the state of a given resource to be lost, whether that state is actually lost may depend on whether the resource is shared with other threads. For example, if a given thread is in a power-saving level that allows loss of the Time Base but shares the Time Base with a thread that is in a power-saving level that does not allow loss of the Time base or is not in power-saving mode, the state of the Time Base must be maintained as if the given thread was not in power-saving mode.

Implementations that do not implement 16 unique levels of power saving may provide the same amount of power saving for more than one power-saving level. However, the resources that are maintained in each power-saving level and the conditions that cause exit from each power-saving level must be as specified by the architecture. In particular, it is always permissible to implement a higher power-saving level as the next-lower level (e.g., to implement power-saving level 0001 as level 0000, or level 1101 as level 1100).

43 **Exit Criterion** (EC)

This field is accessible only to the hypervisor.

0 Hardware will exit power-saving mode when the exception corresponding to any system-caused interrupt occurs.
Power-saving mode is exited either at the instruction following the stop (if MSR_{EE}=0) or in the corresponding interrupt handler (if MSR_{EE}=1).

1 If $SMFCTRL_{UDEE}$ was set to 1 when ***stop*** was executed and $SMFCTRL_{UDEE}$ was not lost, hardware will exit power-saving mode when a Directed Ultravisor Doorbell exception occurs.

If $LPCR_{PECE}$ is not lost, hardware will exit power-saving mode when a System Reset exception or one of the events specified in $LPCR_{PECE}$ occurs. If the event is a Machine Check exception, then a Machine Check interrupt occurs; otherwise a System Reset interrupt occurs, and the contents of SRR1 indicate the event that caused exit from power-saving mode.

When the ***stop*** instruction is executed in hypervisor state, the hypervisor must set the ESL field to the same value as this field. Also, if the RL or MTL fields are set to values that allow state loss, then fields ESL and EC must both be set to 1. Other combinations of the values of the ESL, EC, RL, and MTL fields are reserved for future use.

Architecture Note

Other combinations of the values of the ESL, EC, RL, and MTL fields may be allowed in a future version of the architecture in order to provide additional functionality.

If this field is set to 1 when the ***stop*** instruction is executed in privileged non-hypervisor state, a Hypervisor Facility Unavailable interrupt occurs. See Section 7.5.26.

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 and MTL are less than or equal to PSLL. See Section 7.5.26 for details.

44:47 **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

54:55 **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 7.5.26 of Book III.

Otherwise, if the value of this field is greater than the value of the RL field, the power-saving level is allowed to increase from the value in the RL field up to the value of this field during power-saving mode.

If this field is less than or equal to the value of the PSLL field when ***stop*** is executed in privileged non-hypervisor state, this field is used to specify the maximum power-saving level that can be reached during power-saving mode provided that the value of this field is greater than the value of the RL field. If this field is less than the Requested Level (RL) field when ***stop*** is executed hardware is not allowed to increase the power-saving level during power-saving mode beyond the value indicated in the RL field.

60:63 **Requested Level** (RL)

This field is used to specify the power-saving level that is to be entered when the ***stop*** instruction is executed.

If the value of this field is greater than the value of the Power-Saving Level Limit (PSLL) field when ***stop*** is executed in privileged non-hypervisor state, a Hypervisor Facility Unavailable interrupt occurs.

Programming Note

The Hypervisor Facility Unavailable interrupt occurs when a privileged non-hypervisor program executes ***stop*** when $PSSCR_{RL} > PSSCR_{PSLL}$ so that the Hypervisor may decide whether or not to allow the requested loss of state to occur.

If the hypervisor decides that some loss of state is acceptable, it may choose to re-execute ***stop*** after either setting $PSSCR_{MTL}$ to a value that causes state loss, or setting both $PSSCR_{RL}$ and $PSSCR_{MTL}$ to values that cause state loss. When the thread exits power-saving mode, the hypervisor can quickly determine whether any resources were actually lost and need to be restored.

4.3 Branch Facility Instructions

4.3.1 System Linkage Instructions

These instructions provide the means by which a program can call upon the system to perform a service, and by which the system can return from performing a service or from processing an interrupt.

The *System Call* instruction is described in Book I, but only at the level required by an application programmer. A complete description of this instruction appears below.

System Call SC-form

sc LEV

17	///	///	//	LEV	//	1	/
0	6	11	16	20	27	30	31

```
SRR0 ←iea CIA + 4
SRR1[33:36 42:47] ← 0
SRR1[0:32 37:41 48:63] ← MSR[0:32 37:41 48:63]
MSR ← new_value (see below)
NIA ← 0x0000_0000_0000_0C00
```

The effective address of the instruction following the *System Call* instruction is placed into SRR0. Bits 0:32, 37:41, and 48:63 of the MSR are placed into the corresponding bits of SRR1, and bits 33:36 and 42:47 of SRR1 are set to zero.

Then a System Call interrupt is generated. The interrupt causes the MSR to be set as described in Section 7.5, “Interrupt Definitions” on page 1201. The setting of the MSR is affected by the contents of the LEV field. LEV values greater than 2 are reserved. Bits 0:4 of the LEV field (instruction bits 20:24) are treated as a reserved field.

The interrupt causes the next instruction to be fetched from effective address 0x0000_0000_0000_0C00.

This instruction is context synchronizing.

Special Registers Altered:

Register	*Field(s)*
SRR0	
SRR1	
MSR	

Programming Note

If LEV=1, the hypervisor is invoked.

If LEV=2 and $SMFCTRL_E$ = 1, the ultravisor is invoked.

If LEV=2 and $SMFCTRL_E$ = 0, the hypervisor is invoked. However, such invocation should be considered a programming error.

Executing this instruction with LEV=1 or LEV=2 is the only way that executing an instruction can cause a transition from non-hypervisor state to hypervisor state on the thread that executed the instruction. Executing this instruction with LEV=2 when $SMFCTRL_E$=1 is the only way that executing an instruction can cause a transition from non-ultravisor state to ultravisor state on the thread that executed the instruction.

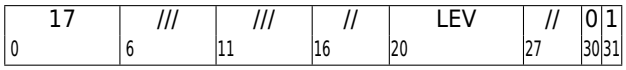
In correct use, this instruction is used to “call up” one privilege level (application program calls operating system, operating system calls hypervisor, hypervisor calls ultravisor). However, it is possible for a program to call up more than one level (e.g., for an application program to call the hypervisor). An attempt to call up more than one level should be considered a programming error.

Programming Note

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

System Call Vectored SC-form

scv LEV




```
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 7.5.

The interrupt causes the next instruction to be fetched as specified in $LPCR_{AIL}$ or $LPCR_{HAIL}$ as appropriate (see Section 2.2).

The SRRs are not affected.

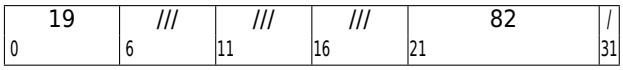
This instruction is context synchronizing.

Special Registers Altered:

Register	*Field(s)*
LR	
CTR	
MSR	

Return From System Call Vectored XL-form

rfscv




```
MSR48 ← CTR48 | CTR49
MSR58 ← (CTR58 | CTR49) & ¬(MSR41 & MSR3 & (¬CTR49))
MSR59 ← (CTR59 | CTR49) & ¬(MSR41 & MSR3 & (¬CTR49))
MSR0:2 4:32 37:40 49:50 52:57 60:63 ←iea CTR0:2 4:32 37:40 49:50 52:57 60:63
NIA ←iea LR0:61 || 0b00
```

The result of ORing bits 48 and 49 of the Count Register is placed into MSR_{48}. The result of ANDing bit 41 of the MSR with bit 3 of the MSR and with the complement of bit 49 of the Count Register is complemented and then ANDed with the result of ORing bits 58 and 49 of the Count Register and placed into MSR_{58}. The result of ANDing bit 41 of the MSR with bit 3 of the MSR and with the complement of bit 49 of the Count Register is complemented and then ANDed with the result of ORing bits 59 and 49 of the Count Register and placed into MSR_{59}. Bits 0:2, 4:32, 37:40, 49:50, 52:57, and 60:63 of the Count Register are placed into the corresponding bits of the MSR.

If the new MSR value does not enable any pending exceptions, then the next instruction is fetched, under control of the new MSR value, from the address $LR_{0:61}$ || 0b00 (when SF=1 in the new MSR value) or $^{32}0$ || $LR_{32:61}$ || 0b00 (when SF=0 in the new MSR value). If the new MSR value enables one or more pending exceptions, the interrupt associated with the highest priority pending exception is generated; in this case the value placed into SRR0, HSRR0, or USRR0 by the interrupt processing mechanism (see Section 7.4.3) is the address of the instruction that would have been executed next had the interrupt not occurred.

This instruction is privileged and context synchronizing.

Special Registers Altered:

Register	*Field(s)*
MSR	

Programming Note

If this instruction sets MSR_{PR} to 1, it also sets MSR_{EE}, MSR_{IR}, and MSR_{DR} to 1. If this instruction results in $MSR_{S\ HV\ PR}$ being equal to 0b110, it also sets MSR_{IR} and MSR_{DR} to 0.

This instruction does not alter MSR_{HV}, MSR_{S}, or MSR_{ME}.

Programming Note

In a future version of the architecture, this instruction may set the contents of the Count Register to zero. For this reason, programs should not rely on its contents after this instruction is executed.

Return From Interrupt Doubleword XL-form

rfid

19	///	///	///	18	/
0	6	11	16	21	31

```
MSR51 ← (MSR3 & SRR151) | ((¬MSR3) & MSR51)
MSR3 ← MSR3 & SRR13
MSR48 ← SRR148 | SRR149
MSR58 ← (SRR158 | SRR149) & ¬(MSR41 & MSR3 & (¬SRR149))
MSR59 ← (SRR159 | SRR149) & ¬(MSR41 & MSR3 & (¬SRR149))
MSR0:2 4:32 37:40 49:50 52:57 60:63
      ← SRR10:2 4:32 37:40 49:50 52:57 60:63
NIA ←iea SRR00:61 || 0b00
```

If MSR_3=1 then bits 3 and 51 of SRR1 are placed into the corresponding bits of the MSR. The result of ORing bits 48 and 49 of SRR1 is placed into MSR_{48}. The result of ANDing bit 41 of the MSR with bit 3 of the MSR and with the complement of bit 49 of SRR1 is complemented and then ANDed with the result of ORing bits 58 and 49 of SRR1 and placed into MSR_{58}. The result of ANDing bit 41 of the MSR with bit 3 of the MSR and with the complement of bit 49 of SRR1 is complemented and then ANDed with the result of ORing bits 59 and 49 of SRR1 and placed into MSR_{59}. Bits 0:2, 4:32, 37:40, 49:50, 52:57, and 60:63 of SRR1 are placed into the corresponding bits of the MSR.

If the new MSR value does not enable any pending exceptions, then the next instruction is fetched, under control of the new MSR value, from the address $SRR0_{0:61}$ || 0b00 (when SF=1 in the new MSR value) or $^{32}0$ || $SRR0_{32:61}$ || 0b00 (when SF=0 in the new MSR value). If the new MSR value enables one or more pending exceptions, the interrupt associated with the highest priority pending exception is generated; in this case the value placed into SRR0, HSRR0, or USRR0 by the interrupt processing mechanism (see Section 7.4.3) is the address of the instruction that would have been executed next had the interrupt not occurred.

This instruction is privileged and context synchronizing.

Special Registers Altered:

Register	*Field(s)*
MSR	

Programming Note

If this instruction sets MSR_{PR} to 1, it also sets MSR_{EE}, MSR_{IR}, and MSR_{DR} to 1. If this instruction results in $MSR_{S\ HV\ PR}$ being equal to 0b110, it also sets MSR_{IR} and MSR_{DR} to 0.

Hypervisor Return From Interrupt Doubleword XL-form

hrfid

19	///	///	///	274	/
0	6	11	16	21	31

```
MSR48 ← HSRR148 | HSRR149
MSR58 ← (HSRR158 | HSRR149)
        & ¬(MSR41 & MSR3 & (¬HSRR149))
MSR59 ← (HSRR159 | HSRR149)
        & ¬(MSR41 & MSR3 & (¬HSRR149))
MSR0:32 37:40 49:57 60:63 ← HSRR10:32 37:40 49:57 60:63
NIA ←iea HSRR00:61 || 0b00
```

The result of ORing bits 48 and 49 of HSRR1 is placed into MSR_{48}. The result of ANDing bit 41 of the MSR with bit 3 of HSRR1 and with the complement of bit 49 of HSRR1 is complemented and then ANDed with the result of ORing bits 58 and 49 of HSRR1 and placed into MSR_{58}. The result of ANDing bit 41 of the MSR with bit 3 of HSRR1 and with the complement of bit 49 of HSRR1 is complemented and then ANDed with the result of ORing bits 59 and 49 of HSRR1 and placed into MSR_{59}. Bits 0:32, 37:40, 49:57, and 60:63 of HSRR1 are placed into the corresponding bits of the MSR.

If the new MSR value does not enable any pending exceptions, then the next instruction is fetched, under control of the new MSR value, from the address $HSRR0_{0:61}$ || 0b00 (when SF=1 in the new MSR value) or $^{32}0$ || $HSRR0_{32:61}$ || 0b00 (when SF=0 in the new MSR value). If the new MSR value enables one or more pending exceptions, the interrupt associated with the highest priority pending exception is generated; in this case the value placed into SRR0, HSRR0, or USRR0 by the interrupt processing mechanism (see Section 7.4.3) is the address of the instruction that would have been executed next had the interrupt not occurred.

This instruction is hypervisor privileged and context synchronizing.

Special Registers Altered:

Register	*Field(s)*
MSR	

Programming Note

If this instruction sets MSR_{PR} to 1, it also sets MSR_{EE}, MSR_{IR}, and MSR_{DR} to 1. If this instruction results in $MSR_{S\ HV\ PR}$ being equal to 0b110, it also sets MSR_{IR} and MSR_{DR} to 0.

Ultravisor Return From Interrupt Doubleword XL-form

urfid

19	///	///	///	306	/
0	6	11	16	21	31

$MSR_{48} \leftarrow USRR1_{48} \mid USRR1_{49}$
$MSR_{58} \leftarrow (USRR1_{58} \mid USRR1_{49})$
$\quad \& \neg(USRR1_{41} \,\&\, USRR1_{3} \,\&\, (\neg USRR1_{49}))$
$MSR_{59} \leftarrow (USRR1_{59} \mid USRR1_{49})$
$\quad \& \neg(USRR1_{41} \,\&\, USRR1_{3} \,\&\, (\neg USRR1_{49}))$
$MSR_{0:32\ 37:41\ 49:57\ 60:63} \leftarrow USRR1_{0:32\ 37:41\ 49:57\ 60:63}$
$NIA \leftarrow_{iea} USRR0_{0:61} \mid\mid 0b00$

The result of ORing bits 48 and 49 of USRR1 is placed into MSR_{48}. The result of ANDing bit 41 of USRR1 with bit 3 of USRR1 and with the complement of bit 49 of USRR1 is complemented and then ANDed with the result of ORing bits 58 and 49 of USRR1 and placed into MSR_{58}. The result of ANDing bit 41 of USRR1 with bit 3 of USRR1 and with the complement of bit 49 of USRR1 is complemented and then ANDed with the result of ORing bits 59 and 49 of USRR1 and placed into MSR_{59}. Bits 0:32, 37:41, 49:57, and 60:63 of USRR1 are placed into the corresponding bits of the MSR.

If the new MSR value does not enable any pending exceptions, then the next instruction is fetched, under control of the new MSR value, from the address $USRR0_{0:61}$ || 0b00 (when SF=1 in the new MSR value) or $^{32}0$ || $USRR0_{32:61}$ || 0b00 (when SF=0 in the new MSR value). If the new MSR value enables one or more pending exceptions, the interrupt associated with the highest priority pending exception is generated; in this case the value placed into SRR0, HSRR0, or USRR0 by the interrupt processing mechanism (see Section 7.4.3) is the address of the instruction that would have been executed next had the interrupt not occurred.

This instruction is ultravisor privileged and context synchronizing.

Special Registers Altered:

Register	*Field(s)*
MSR	

Programming Note

If this instruction sets MSR_{PR} to 1, it also sets MSR_{EE}, MSR_{IR}, and MSR_{DR} to 1. If this instruction sets $MSR_{S\ HV\ PR}$ to 0b110, it also sets MSR_{IR} and MSR_{DR} to 0.

4.3.2 Power-Saving Mode

Power-Saving Mode is a mode in which the thread does not execute instructions and may consume less power than it would if it were not in power-saving mode.

There are 16 levels of power savings, designated as levels 0-15. For each power-saving level, the power consumed may be less than or equal to the power consumed in the next-lower level, and the time required for the thread to exit power-saving mode and resume execution may be greater than or equal that of the next-lower level.

When the thread is in power-saving mode, some resource state may be lost. The state that may be lost while in each power-saving level is implementation dependent, with the following restrictions.

- For $PSSCR_{ESL}$ = 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, $SMFCTRL_E$, and the URMOR in an SMF-enabled system or the HRMOR in an SMF-disabled system is always maintained.

Programming Note

For the power-saving level corresponding to the second item above, if the state of the Decrementer were not maintained and updated as if the thread was not in power-saving mode, Decrementer exceptions would not reliably cause exit from this power-saving level even if Decrementer exceptions were enabled to cause exit.

Programming Note

The first two items above provide the equivalent of ***doze*** and ***nap*** in Power ISA V 2.07, respectively. Higher power-saving levels provide the equivalent of ***sleep*** and ***rvwinkle***.

Engineering Note

Certain bits in certain implementation-specific registers (e.g., “HID” registers) may need to be preserved in power-saving levels that allow loss of state.

Engineering Note

Before power can be removed from a cache, all modified data must be written to storage.

Engineering Note

The actions required by the program in order to resume normal operation after the thread exits from power-saving mode should not require synchronization with other threads that do not share instruction execution facilities with the thread exiting power saving mode. (Requiring synchronization with one other entity, such as a “Service Processor”, is acceptable.)

Engineering Note

The sequential execution model requires that all instructions preceding the ***stop*** instruction appear to complete before the power saving mode is entered. If a previous move to the Time Base, Decrementer, Hypervisor Decrementer, or PSSCR incorrectly completed after the power-saving mode had been entered, the resulting register value could be noticeably different than if the move completed correctly before the power-saving mode had been entered.

4.3.2.1 Power-Saving Mode Instruction

The ***stop*** instruction is used to stop instruction fetching and execution and put the thread into power-saving mode. The thread remains in power-saving mode until a system reset exception or an event that is enabled to cause exit from power-saving mode occurs. (See the definition of $PSSCR_{EC}$ in Section 4.2.2.)

Stop XL-form

stop

19	///	///	///	370	/
0	6	11	16	21	31

The thread is placed into power-saving mode and execution is stopped.

The power-saving level that is entered is determined by the contents of the PSSCR (see Section 4.2.2). The thread state that is maintained depends on the power-saving level that is entered. The thread state that is maintained at each power-saving level is implementation-dependent, subject to the restrictions specified in Section 4.3.2.MSR_{EE}=0) or in the corresponding interrupt handler (if MSR_{EE}=1).

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 $PSSCR_{EC}$, $LPCR_{PECE}$, and $SMFCTRL_{UDEE}$. 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.

This instruction should not be executed in ultravisor state because that scenario may not be thoroughly verified.

This instruction is privileged and context synchronizing.

Special Registers Altered:

None

Programming Note

The result of executing a *Stop* instruction is determined by the value of $MSR_{S\ HV\ PR}$ as follows.

xx1 Privileged Instruction type Program Interrupt

000 state loss not allowed: execute ***stop***

000 state loss allowed: Hypervisor Facility Unavailable interrupt to the hypervisor

010 execute ***stop***

100 state loss not allowed: execute ***stop***

100 state loss allowed: Hypervisor Facility Unavailable interrupt to the ultravisor

110 execute ***stop*** (ultravisor programming error)

4.3.2.2 Entering and Exiting Power-Saving Mode

Before software executes the ***stop*** instruction, the PSSCR is initialized. If the ***stop*** instruction is to be used by the OS, the hypervisor initializes the fields that are accessible only to the hypervisor before dispatching the OS. These fields include the SD, ESL, EC, and PSLL fields. See the Programming Notes for these fields in Section 4.2.2 for additional information.

If the ***stop*** instruction is to be executed by the hypervisor when $PSSCR_{EC}$=1, $LPCR_{PECE}$ and $SMFCTRL_{UDEE}$ must be set to the desired value (see Sections 2.2 and 3.4). Depending on the implementation and the power-saving level to be entered, it may also be necessary to save the state of certain resources and perform synchronization procedures to ensure that all stores have been performed with respect to other threads or mechanisms that use the storage areas before executing the ***stop***. See the the User's Manual for the implementation for details.

Software must also specify the requested and maximum power-saving level limit fields (i.e RL and MTL fields), and the Transition Rate (TR) field in the PSSCR in order to bound the range of power-saving modes that can be entered. If the value of the RL field is greater than or equal to the value of the MTL field, the power-saving level will not increase from the initial level during power-saving mode.

Programming Note

If MSR_{EE}=1 when the ***stop*** instruction is executed, then the interrupt corresponding to the exception that was expected to cause exit from power-saving mode may occur immediately prior to execution of the ***stop*** instruction. If this occurs, the result may be a software hang condition since the exception that was expected to cause exit from power-saving mode has already occurred.

The above software hang condition can be prevented by setting MSR_{EE}=0 prior to executing ***stop***.

After the thread has entered power-saving mode with $PSSCR_{EC}$=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 MSR_{EE}=0) or in the corresponding interrupt handler (if MSR_{EE}=1).

Programming Note

If ***stop*** was executed when $PSSCR_{EC}$=0, then $PSSCR_{ESL}$ must also be set to 0 and $PSSCR_{RL\ MTL}$ must be set to values that do not allow state loss. (See the definition of the EC bit description in Section 4.3.2.) This guarantees that the state of MSR_{EE} is not lost.

Programming Note

If ***stop*** was executed when $PSSCR_{EC}$=0 and MSR_{EE}=0 (in order to avoid the hang condition described in a preceding Programming Note), MSR_{EE} 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.

After the thread has entered power-saving mode with $PSSCR_{EC}$=1, only the System Reset exception and the exceptions enabled in $LPCR_{PECE}$ and $SMFCTRL_{UDEE}$ will cause exit. If the event that causes exit is a Machine Check exception, then a Machine Check interrupt occurs; otherwise a System Reset interrupt occurs, and the contents of SRR1 indicate the exception that caused exit from power-saving mode. If state loss has occurred in an SMF-enabled system, the interrupt is taken in ultravisor state.

If the hypervisor has set $PSSCR_{SD}$=0 prior to when the ***stop*** instruction is executed, the instruction following the ***stop*** may typically be a ***mfspr*** in order to read the contents of $PSSCR_{PLS}$ to determine the maximum power-saving level that was entered during power-saving mode.

Programming Note

The ultravisor does not initiate power-saving.

If a secure partition attempts to execute ***stop*** with parameters that allow state loss, the ultravisor gets control via the Hypervisor Facility Unavailable interrupt. It saves secure state and gives control to the hypervisor's Hypervisor Facility Unavailable interrupt handler.

Upon exit from a state-losing power-saving mode in an SMF-enabled system, the ultravisor gets control at its Machine Check or System Reset interrupt handler. It restores any ultravisor state that was lost, and then services the Directed Ultravisor Doorbell exception if that caused the wakeup. It then restores the HRMOR and transfers control to the hypervisor at the hypervisor's Machine Check interrupt handler if the ultravisor got control at the ultravisor's Machine Check interrupt handler, and to the hypervisor's System Reset interrupt handler otherwise.. The hypervisor restores any lost hypervisor state, and then handles the exception (other than Directed Ultravisor Doorbell exception) that caused the wakeup. For this process to work, the ultravisor must have stored a record of its state in some known location prior to transferring control to the hypervisor to execute ***stop***. The hypervisor in turn must have stored its HRMOR value in a location known to the ultravisor. It must also have stored a record of its state in some known location.

The only other function the ultravisor may need to perform for a given power-saving mode transition is to be a proxy accessing hypervisor state in the platform that is mixed with ultravisor state and lacking independent access control.

4.4 Event-Based Branch Facility and Instruction

The Event-Based Branch facility is described in Chapter 6 of Book II, but only at the level required by the application program.

Event-based branches can only occur in problem state and when event-based branches and exceptions have been enabled in the FSCR and HFSCR, and $BESCR_{GE}$=1. Additionally, the following additional bits must be set to one in order to enable EBB exceptions specific to a given function to occur.

- $MMCR0_{EBE}$ and $BESCR_{PME}$ must be set to 1 to enable Performance Monitor event-based exceptions.
- $BESCR_{EE}$ must be set to 1 to enable External event-based exceptions.

If an event-based exception exists (as indicated by $BESCR_{PMEO}$=1 or $BESCR_{EEO}$=1) when MSR_{PR}=0, the corresponding event-based branch will occur when MSR_{PR}=1, $FSCR_{EBB}$=1, $HFSCR_{EBB}$=1, and $BESCR_{GE}$=1.

Programming Note

Software EBB handlers should ensure that previous exceptions have been cleared (by setting $BESCR_{PMEO}$ and/or $BESCR_{EEO}$ to 0) before re-enabling event-based branches (by setting $BESCR_{GE}$ to 1 or executing rfebb 1) in order to prevent earlier exceptions from causing additional EBBs.

Chapter 5. Fixed-Point Facility

5.1 Fixed-Point Facility Overview

This chapter describes the details concerning the registers and the privileged instructions implemented in the Fixed-Point Facility that are not covered in Book I.

5.2 Special Purpose Registers

Special Purpose Registers (SPRs) are read and written using the ***mfspr*** (page 1107) and ***mtspr*** (page 1105) instructions. Most SPRs are defined in other chapters of this book; see the index to locate those definitions.

5.3 Fixed-Point Facility Registers

5.3.1 Processor Version Register

The Processor Version Register (PVR) is a 32-bit read-only register that contains a value identifying the version and revision level of the implementation. The contents of the PVR can be copied to a GPR by the ***mfspr*** instruction. Read access to the PVR is privileged; write access is not provided.

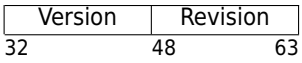


Figure 5.1: Processor Version Register

The PVR distinguishes between implementations that differ in attributes that may affect software. It contains two fields.

Version A 16-bit number that identifies the version of the implementation. Different version numbers indicate major differences between implementations.

Revision A 16-bit number that distinguishes between implementations of the version. Different revision numbers indicate minor differences between implementations having the same version number, such as clock rate and Engineering Change level.

Version numbers are assigned by the Power ISA process. Revision numbers are assigned by an implementation-defined process.

Engineering Note

Although the classification of a given difference between implementations as “major” or “minor” is somewhat arbitrary, the following are examples of differences that generally should be considered “major”.

- number and types of execution units
- level of support of instructions (hard-wired or emulated)
- size, geometry, and management of caches and of TLBs

The following are examples of differences that generally should be considered “minor”.

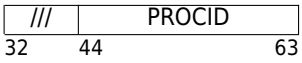
- remapping a implementation to a new technology
- redesigning a critical path to increase clock rate
- fixing bugs

In general, any change to an implementation should cause a new PVR value to be assigned. Even a seemingly trivial change that is not expected to be apparent to software should cause a new revision number to be assigned, in case the change is later discovered to have introduced an error that software must circumvent.

[]

5.3.2 Processor Identification Register

The Processor Identification Register (PIR) is a 32-bit register that contains a 20-bit PROCID field that can be used to distinguish the thread from other threads in the system. The contents of the PIR can be copied to a GPR by the ***mfspr*** instruction. Read access to the PIR is privileged; write access is not provided.



Bits	Name	Description
32:43		Reserved
44:63	PROCID	Thread ID

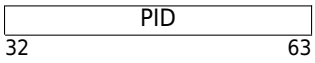
Figure 5.2: Processor Identification Register

The means by which the PIR is initialized are implementation-dependent.

The PIR is a hypervisor resource; see Chapter 2.

5.3.3 Process Identification Register

The layout of the Process Identification Register (PIDR) is shown in Figure 5.3 below.



Bit(s)	Name	Description
32:63	PID	Process Identifier

Figure 5.3: Process Identification Register

The contents of the PIDR identify the process to which the thread is assigned. The value is used to perform translation and manage the caching of translations. The number of PIDR bits supported is implementation-dependent.

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 HR=1, PIDR should not be set to zero except when MSR_{PR}=0.

[]

5.3.4 Control Register

The Control Register (CTRL) is a 32-bit register as shown below.

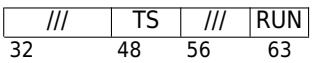


Figure 5.4: Control Register

The field definitions for the CTRL are shown below.

Bit(s) Definition

32:47 Reserved

48:55 **Thread State** (TS)

Problem State Access
Reserved

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.

Architecture Note

Future versions of the Architecture may extend the TS field to occupy bits 48:62.

The current definition of the TS field allows reporting of the state of the RUN bit only for threads with privileged or hypervisor thread numbers 0:7. If an implementation supports additional threads, the state of the RUN bit for threads with privileged or hypervisor thread numbers 0:7 is still required to be reported in the TS field. For such implementations, the state of the RUN bit for threads with higher privileged or hypervisor thread numbers is not reported in the TS field.

56:62 Reserved

63 **Run** (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 11)

The RUN bit can be used by the operating system to indicate when the thread is doing useful work.

Write access to the CTRL is privileged. Reads can be performed in privileged or problem state.

5.3.5 Program Priority Register

Privileged programs may set a wider range of program priorities in the PRI field of PPR and PPR32 than may be set by problem state programs (see Chapter 3 of Book II). Problem state programs may only set values in the range of 0b001 to 0b100 unless the Problem State Priority Boost register (see Section 5.3.6) allows the value 0b101. Privileged programs may set values in the range of 0b001 to 0b110. Hypervisor software may also set 0b111. For all priorities except 0b101, if a program attempts to set a value that is not allowed for its privilege level, the PRI field remains unchanged. If a problem state program attempts to set its priority value to 0b101 when this priority value is not allowed for problem state programs, the priority is set to 0b100. The values and their corresponding meanings are as follows.

Bit(s) Definition

11:13 ***Program Priority*** (PRI)

001	very low
010	low
011	medium low
100	medium
101	medium high
110	high
111	very high

5.3.6 Problem State Priority Boost Register

The Problem State Priority Boost (PSPB) register is a 32-bit register that controls whether problem state programs have access to program priority medium high. (See Section 3.1 of Book II.)

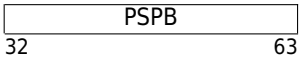


Figure 5.5: Problem State Priority Boost Register

A problem state program is able to set the program priority to medium high only when the PSPB of the thread contains a non-zero value.

The maximum value to which the PSPB can be set must be a power of 2 minus 1. Bits that are not required to represent this maximum value must return 0s when read regardless of what was written to them.

When the PSPB is set to a value less than its maximum value but greater than 0, its contents decrease monotonically at the same rate as the SPURR until its contents minus the amount it is to be decreased are 0 or less when a problem state program is executing on the thread at a priority of medium high.When the contents of the PSPB minus the amount it is to be decreased are 0 or less, its contents are replaced by 0.

When the PSPB is set to its maximum value or 0, its contents do not change until it is set to a different value.

Whenever the priority of a thread is medium high and either of the following conditions exist, hardware changes the priority to medium:

- the PSPB counts down to 0, or
- PSPB=0 and the privilege state of the thread is changed to problem state (MSR_{PR}=1).

5.3.7 Relative Priority Register

The Relative Priority Register (RPR) is a 64-bit register that allows the hypervisor to control the relative priorities corresponding to each valid value of PPR_{PRI}.

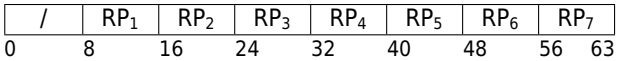


Figure 5.6: Relative Priority Register

Each RP_n field is defined as follows.

Bit(s) Definition

0:1 Reserved

2:7 **Relative priority of priority level n**

Specifies the relative priority that corresponds to the priority corresponding to PPR_{PRI}=n, where a value of 0 indicates the lowest relative priority and a value of 0b111111 indicates the highest relative priority.

Programming Note

The hypervisor must ensure that the values of the RP_n fields increase monotonically for each n and are of different enough magnitudes to ensure that each priority level provides a meaningful difference in priority.

5.3.8 Hash Key Registers

The Hash Key Register (HASHKEYR) is a 64-bit register which contains the key for the hash computation done by the non-privileged *Hash* instructions, namely ***hashst*** and ***hashchk*** (see Section 3.3.17 of Book I). Similarly, the Hash Privileged Key Register (HASHPKEYR) is a 64-bit register which contains the key for the hash computation done by the privileged *Hash* instructions, namely ***hashstp*** and ***hashchkp*** (see Section 5.4.2 of Book III).

Figure 5.7: Hash Key Register

Figure 5.8: Hash Privileged Key Register

HASHKEYR is a privileged register. HASHPKEYR is a hypervisor privileged register.

Programming Note

Normally any indication of privilege in the name and mnemonic of an SPR appears at the beginning – e.g., Hypervisor Decrementer (HDEC), Ultravisor Real Mode Offset Register (URMOR). The Hash Privileged Key Register (HASHPKEYR) seems to violate this convention, because the privilege indication is in the middle of the name and mnemonic. But “Privileged (P)” for this SPR indicates the privilege not of the SPR – which is hypervisor privileged – but of the instructions (***hashstp*** and ***hashchkp***) that use the SPR. So the seeming violation of the convention is appropriate.

5.3.9 Software-use SPRs

Software-use SPRs are 64-bit registers provided for use by software.

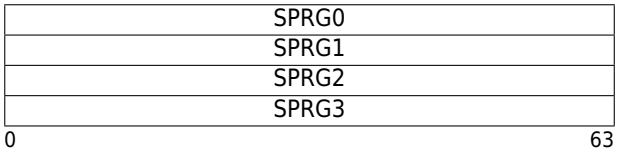


Figure 5.9: Software-use SPRs

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 ***mfspr*** instruction.

Programming Note

Neither the contents of the SPRGs, nor accessing them using ***mtspr*** or ***mfspr***, has a side effect on the operation of the thread. One or more of the registers is likely to be needed by interrupt handlers that run in privileged non-hypervisor state (e.g., as scratch registers and/or pointers to per thread save areas).

Operating systems must ensure that no sensitive data are left in SPRG3 when a problem state program is dispatched, and operating systems for secure systems must ensure that SPRG3 cannot be used to implement a “covert channel” between problem state programs. These requirements can be satisfied by clearing SPRG3 before passing control to a program that will run in problem state.

HSPRG0 and HSPRG1 are 64-bit registers provided for use by hypervisor programs.

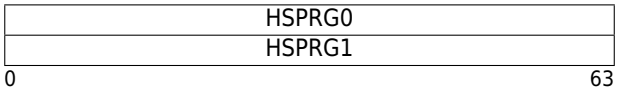


Figure 5.10: SPRs for use by hypervisor programs

Programming Note

Neither the contents of the HSPRGs, nor accessing them using ***mtspr*** or ***mfspr***, has a side effect on the operation of the thread. One or both of the registers is likely to be needed by interrupt handlers that run in hypervisor non-ultravisor state (e.g., as scratch registers and/or pointers to per thread save areas).

USPRG0 and USPRG1 are 64-bit registers provided for use by ultravisor programs.

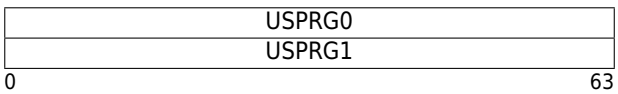


Figure 5.11: SPRs for use by ultravisor programs

Programming Note

Neither the contents of the USPRGs, nor accessing them using ***mtspr*** or ***mfspr***, has a side effect on the operation of the thread. One or both of the registers is likely to be needed by interrupt handlers that run in ultravisor state (e.g., as scratch registers and/or pointers to per thread save areas).

5.4 Fixed-Point Facility Instructions

5.4.1 Fixed-Point Load and Store Caching Inhibited Instructions

The storage accesses caused by the instructions described in this section are performed as though the specified storage location is Caching Inhibited and Guarded. The instructions can be executed only in hypervisor state. Software must ensure that the specified storage location is not in the caches. If the specified storage location is in a cache, the results are undefined.

The *Fixed-Point Load and Store Caching Inhibited* instructions must be executed only when MSR_{DR}=0. The storage location specified by the instructions must not be in storage specified by the Hypervisor Real Mode Storage Control facility to be treated as non-Guarded. If either of these conditions is violated, the result is a Data Storage interrupt.

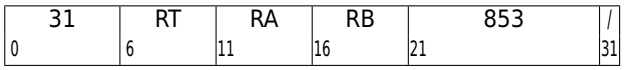
Programming Note

The instructions described in this section can be used to permit a control register on an I/O device to be accessed without permitting the corresponding storage location to be copied into the caches.

The *Fixed-Point Load and Store Caching Inhibited* instructions are fixed-point *Storage Access* instructions; see Section 3.3.1 of Book I.

Load Byte and Zero Caching Inhibited Indexed X-form

lbzcix RT,RA,RB




```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + (RB)
RT ← 56 0 || MEM(EA, 1)
```

Let the effective address (EA) be the sum (RA|0)+ (RB). The byte in storage addressed by EA is loaded into $RT_{56:63}$. $RT_{0:55}$ are set to 0.

The storage access caused by this instruction is performed as though the specified storage location is Caching Inhibited and Guarded.

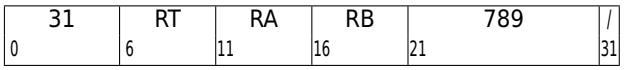
This instruction is hypervisor privileged.

Special Registers Altered:

None

Load Word and Zero Caching Inhibited Indexed X-form

lwzcix RT,RA,RB




```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + (RB)
RT ← 32 0 || MEM(EA, 4)
```

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

The storage access caused by this instruction is performed as though the specified storage location is Caching Inhibited and Guarded.

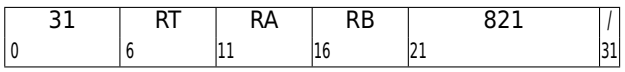
This instruction is hypervisor privileged.

Special Registers Altered:

None

Load Halfword and Zero Caching Inhibited Indexed X-form

lhzcix RT,RA,RB




```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + (RB)
RT ← 48 0 || MEM(EA, 2)
```

Let the effective address (EA) be the sum (RA|0)+ (RB). The halfword in storage addressed by EA is loaded into $RT_{48:63}$. $RT_{0:47}$ are set to 0.

The storage access caused by this instruction is performed as though the specified storage location is Caching Inhibited and Guarded.

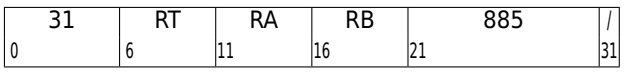
This instruction is hypervisor privileged.

Special Registers Altered:

None

Load Doubleword Caching Inhibited Indexed X-form

ldcix RT,RA,RB




```
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.

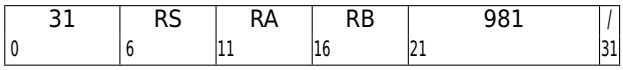
This instruction is hypervisor privileged.

Special Registers Altered:

None

Store Byte Caching Inhibited Indexed X-form

stbcix RS,RA,RB




```
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.

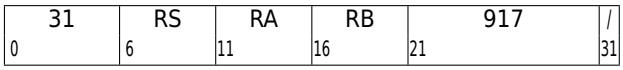
This instruction is hypervisor privileged.

Special Registers Altered:

None

Store Word Caching Inhibited Indexed X-form

stwcix RS,RA,RB




```
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.

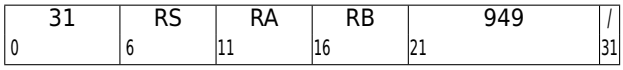
This instruction is hypervisor privileged.

Special Registers Altered:

None

Store Halfword Caching Inhibited Indexed X-form

sthcix RS,RA,RB




```
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.

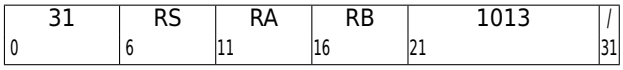
This instruction is hypervisor privileged.

Special Registers Altered:

None

Store Doubleword Caching Inhibited Indexed X-form

stdcix RS,RA,RB




```
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

5.4.2 Fixed-Point Hash Instructions

Privileged instructions analogous to the *Hash* instructions described in Section 3.3.17 of Book I are defined in this section. The HashDigest function, used below, is defined in Section 3.3.17.1 of Book I.

Hash Store Privileged X-form

hashstp RB,offset(RA)

31	D	RA	RB	658	DX
0	6	11	16	21	31

```
DW ← 32×DX + D
d ← EXTS(0b111_1111 || DW || 0b000)
EA ← (RA) + d
temp ← HashDigest((RA),(RB),(HASHPKEYR))
MEM(EA, 8) ← temp
```

Let DW be the value 32xDX + D. The offset is (0b111_1111 || DW || 0b000) sign extended to 64 bits. Let the effective address (EA) be the sum (RA) + offset. The doubleword hash value computed from the contents of RA, RB, and the hypervisor privileged SPR HASHPKEYR, as specified by the HashDigest function described in Section 3.3.17.1 of Book I, is stored into the doubleword in storage addressed by EA.

This instruction is privileged.

EA must be a multiple of 8. If it is not, either an Alignment interrupt occurs or the results are boundedly undefined.

If RA=0, the instruction form is invalid.

Special Registers Altered:

None

Hash Check Privileged X-form

hashchkp RB,offset(RA)

31	D	RA	RB	690	DX
0	6	11	16	21	31

```
DW ← 32×DX + D
d ← EXTS(0b111_1111 || DW || 0b000)
EA ← (RA) + d
temp ← HashDigest((RA),(RB),(HASHPKEYR))
temp1 ← MEM(EA, 8)
if (temp ≠ temp1) then TRAP
```

Let DW be the value 32xDX + D. The offset is (0b111_1111 || DW || 0b000) sign extended to 64 bits. Let the effective address (EA) be the sum (RA) + offset. The doubleword in storage addressed by EA is read and compared with the doubleword hash value computed from the contents of RA, RB, and the hypervisor privileged SPR HASHPKEYR, as specified by the HashDigest function described in Section 3.3.17.1 of Book I. If the values are unequal, the system trap handler is invoked.

This instruction is treated as a *Load*; see Section 4.3 of Book II.

This instruction is privileged.

If the values are unequal, this instruction is context synchronizing (see Book III).

EA must be a multiple of 8. If it is not, either an Alignment interrupt occurs or the results are boundedly undefined.

If RA=0, the instruction form is invalid.

Special Registers Altered:

None

Programming Note

See the Programming Notes that appear in the description of ***hashst*** and ***hashchk*** in Section 3.3.17.2 of Book I.

5.4.3 OR Instruction

or Rx,Rx,Rx can be used to set PPR_{PRI} (see Section 5.3.5) as shown in Figure 5.12. 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.)

Rx	PPR_{PRI}	Priority	Privileged
31	001	very low	no
1	010	low	no
6	011	medium low	no
2	100	medium	no
5	101	medium high	no/yes[1]
3	110	high	yes
7	111	very high	hypv
[1]This value is privileged unless the Problem State Priority Boost register allows the priority value 0b101 (See Section 5.3.6.)			

Figure 5.12: Priority levels for ***or*** *Rx,Rx,Rx*

Architecture Note

or *Rx,Rx,Rx* having an Rx value that is not shown in Figure 5.12 or used for a cache control hint (see Section 4.3.3 of Book II) may be used by some implementations for implementation-specific purposes. These forms of the *or Rx,Rx,Rx* should not be assigned a meaning in the Power ISA except after careful consideration of the effect of such assignment on existing implementations.

[]

5.4.4 Move To/From System Register Instructions

The *Move To Special Purpose Register* and *Move From Special Purpose Register* instructions are described in Book I, but only at the level available to an application programmer. For example, no mention is made there of registers that can be accessed only in privileged state. The descriptions of these instructions given below extend the descriptions given in Book I, but do not list Special Purpose Registers that are implementation-dependent. In the descriptions of these instructions given in below, the "defined" SPR numbers are the SPR numbers shown in the Table 5.1 for the instruction and the implementation-specific SPR numbers that are implemented, and similarly for "defined" registers. All other SPR numbers are undefined for the instruction. (Implementation-specific SPR numbers that are not implemented are considered to be undefined.) When an SPR is defined for ***mtspr*** and undefined for ***mfspr***, or vice versa, a hyphen appears in the column for the instruction for which the SPR number is undefined.

SPR numbers that are not shown in Table 5.1 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.

Architecture Note

SPR numbers that are in the ranges 28-29, 80-82, 136-142, 144-159, 276-279, 512-639, and 972-973 are used in some early implementations for implementation-specific purposes. These SPR numbers will not be assigned a meaning in the Power ISA except after careful consideration of the effect of such assignment on existing implementations.

Architecture Note

In Versions 3.0 and 3.0B of the architecture, SPR 158 was defined to be the GSR (Group Start Register). This SPR number will not be assigned a meaning in the Power ISA except after careful consideration of the possibility that existing (privileged) software might contain *mtspr 158,Rx*.

Architecture Note

SPR numbers 128-131 will be among the last to be assigned a meaning. In earlier versions of the architecture they were used by Transactional Memory SPRs TFHAR, TFIAR, TEXASR, and TEXASRU.

Table 5.1: SPR encodings

decimal	SPR[1]	Register Name	Privileged		Length	Extended Mnemonics*	
	$spr_{5:9}$ $spr_{0:4}$		mtspr	mfspr	(bits)	mtspr	mfspr
1	00000 00001	XER	no	no	64	mtxer Rx	mfxer Rx
3	00000 00011	DSCR	no	no	64	mtudscr	mfudscr
8	00000 01000	LR	no	no	64	mtlr Rx	mflr Rx
9	00000 01001	CTR	no	no	64	mtctr Rx	mfctr Rx
13	00000 01101	AMR	no[4]	no	64	mtuamr Rx	mfuamr Rx
17	00000 10001	DSCR	yes	yes	64	mtdscr Rx	mfdscr Rx
18	00000 10010	DSISR	yes	yes	32	mtdsisr Rx	mfdsisr Rx
19	00000 10011	DAR	yes	yes	64	mtdar Rx	mfdar Rx
22	00000 10110	DEC	yes	yes	64	mtdec Rx	mfdec Rx
26	00000 11010	SRR0	yes	yes	64	mtsrr0 Rx	mfsrr0 Rx
27	00000 11011	SRR1	yes	yes	64	mtsrr1 Rx	mfsrr1 Rx
28	00000 11100	CFAR	yes	yes	64	mtcfar Rx	mfcfar Rx
29	00000 11101	AMR	yes[4]	yes	64	mtamr Rx	mfamr Rx
48	00001 10000	PIDR	yes	yes	32	mtpidr Rx	mfpidr Rx
61	00001 11101	IAMR	yes[7]	yes	64	mtiamr Rx	mfiamr Rx
136	00100 01000	CTRL	-	no	32	-	mfctrl Rx
[]							
152	00100 11000	CTRL	yes	-	32	mtctrl Rx	-
153	00100 11001	FSCR	yes	yes	64	mtfscr Rx	mffscr Rx
157	00100 11101	UAMOR	yes[5]	yes	64	mtuamor Rx	mfuamor Rx
158	00100 11110	na	yes	-	na	-	-
159	00100 11111	PSPB	yes	yes	32	mtpspb Rx	mfpspb Rx
176	00101 10000	DPDES	hypv[2]	yes	64	mtdpdes Rx	mfdpdes Rx
180	00101 10100	DAWR0	hyp/ult[13]	hyp/ult[13]	64	mtdawr0 Rx	mfdawr0 Rx
181	00101 10101	DAWR1	hyp/ult[13]	hyp/ult[13]	64	mtdawr1 Rx	mfdawr1 Rx
186	00101 11010	RPR	hypv[2]	hypv[2]	64	mtrpr Rx	mfrpr Rx
187	00101 11011	CIABR	hyp/ult[13]	hyp/ult[13]	64	mtciabr Rx	mfciabr Rx
188	00101 11100	DAWRX0	hyp/ult[13]	hyp/ult[13]	32	mtdawrx0 Rx	mfdawrx0 Rx
189	00101 11101	DAWRX1	hyp/ult[13]	hyp/ult[13]	32	mtdawrx1 Rx	mfdawrx1 Rx
190	00101 11110	HFSCR	hypv[2]	hypv[2]	64	mthfscr Rx	mfhfscr Rx
256	01000 00000	VRSAVE	no	no	32	mtvrsave Rx	mfvrsave Rx
259	01000 00011	SPRG3	-	no	64	-	mfusprg3
268	01000 01100	TB	-	no	64	-	mftb Rx[10]
269	01000 01101	TBU	-	no	32	-	mftbu Rx[10]
272-275	01000 100xx	SPRG[n] n=0-3	yes	yes	64	mtspgrn Rx	mfspgrn Rx
[]							
284	01000 11100	TBL	hypv[2]	-	32	mttbl Rx	-
285	01000 11101	TBU	hypv[2]	-	32	mttbu Rx	-
286	01000 11110	TBU40	hypv	-	64	mttbu40 Rx	-
287	01000 11111	PVR	-	yes	32	-	mfpvr Rx
304	01001 10000	HSPRG0	hypv[2]	hypv[2]	64	mthsprg0 Rx	mfhsprg0 Rx
305	01001 10001	HSPRG1	hypv[2]	hypv[2]	64	mthsprg1 Rx	mfhsprg1 Rx
306	01001 10010	HDSISR	hypv[2]	hypv[2]	32	mthdisr Rx	mfhdisr Rx
307	01001 10011	HDAR	hypv[2]	hypv[2]	64	mthdar Rx	mfhdar Rx
308	01001 10100	SPURR	hypv[2]	yes	64	mtspurr Rx	mfspurr Rx
309	01001 10101	PURR	hypv[2]	yes	64	mtpurr Rx	mfpurr Rx
310	01001 10110	HDEC	hypv[2]	hypv[2]	64	mthdec Rx	mfhdec Rx
313	01001 11001	HRMOR	hypv[2]	hypv[2]	64	mthrmor Rx	mfhrmor Rx

Table 5.1: SPR encodings (continued)

decimal	SPR[1]	Register Name	Privileged		Length	Extended Mnemonics*	
	$spr_{5:9}$ $spr_{0:4}$		mtspr	mfspr	(bits)	mtspr	mfspr
314	01001 11010	HSRR0	hypv[2]	hypv[2]	64	mthsrr0 Rx	mfhsrr0 Rx
315	01001 11011	HSRR1	hypv[2]	hypv[2]	64	mthsrr1 Rx	mfhsrr1 Rx
318	01001 11110	LPCR	hypv[2]	hypv[2]	64	mtlpcr Rx	mflpcr Rx
319	01001 11111	LPIDR	hypv[2]	hypv[2]	32	mtlpidr Rx	mflpidr Rx
336	01010 10000	HMER	hypv[2,3]	hypv[2]	64	mthmer Rx	mfhmer Rx
337	01010 10001	HMEER	hypv[2]	hypv[2]	64	mthmeer Rx	mfhmeer Rx
338	01010 10010	PCR	hypv[2]	hypv[2]	64	mtpcr Rx	mfpcr Rx
339	01010 10011	HEIR	hypv[2]	hypv[2]	64	mtheir Rx	mfheir Rx
349	01010 11101	AMOR	hypv[2]	hypv[2]	64	mtamor Rx	mfamor Rx
446	01101 11110	TIR	-	yes	64	-	mftir Rx
455	01110 00111	HDEXCR	-	no	32	-	mfuhdexcr Rx
464	01110 10000	PTCR	hypv[14]	hypv[2]	64	mtptcr Rx	mfptcr Rx
468	01110 10100	HASHKEYR	yes	yes	64	mthashkeyr Rx	mfhashkeyr Rx
469	01110 10101	HASHPKEYR	hypv[2]	hypv[2]	64	mthashpkeyr Rx	mfhashpkeyr Rx
471	01110 10111	HDEXCR	hypv[2]	hypv[2]	64	mthdexcr Rx	mfhdexcr Rx
496	01111 10000	USPRG0	ultv	ultv	64	mtusprg0 Rx	mfusprg0 Rx
497	01111 10001	USPRG1	ultv	ultv	64	mtusprg1 Rx	mfusprg1 Rx
505	01111 11001	URMOR	ultv	ultv	64	mturmor Rx	mfurmor Rx
506	01111 11010	USRR0	ultv	ultv	64	mtusrr0 Rx	mfusrr0 Rx
507	01111 11011	USRR1	ultv	ultv	64	mtusrr1 Rx	mfusrr1 Rx
511	01111 11111	SMFCTRL	ultv	ultv	64	mtsmfctrl Rx	mfsmfctrl Rx
736	10111 00000	SIER2	-	no[6]	64	-	mfusier2 Rx mfsier2 Rx
737	10111 00001	SIER3	-	no[6]	64	-	mfusier3 Rx mfsier3 Rx
738	10111 00010	MMCR3	-	no[6]	64	-	mfummcr3 Rx mfmmcr3 Rx
752	10111 10000	SIER2	yes	yes	64	mtsier2 Rx	[12]
753	10111 10001	SIER3	yes	yes	64	mtsier3 Rx	[12]
754	10111 10010	MMCR3	yes	yes	64	mtmmcr3 Rx	[12]
768	11000 00000	SIER	-	no[6]	64	-	mfusier Rx mfsier Rx
769	11000 00001	MMCR2	no[6]	no[6]	64	mtummcr2 Rx mtmmcr2 Rx	mfummcr2 Rx mfmmcr2 Rx
770	11000 00010	MMCRA	no[6]	no[6]	64	mtummcra Rx	mfummcra Rx mfmmcra Rx
771	11000 00011	PMC1	no[6]	no[6]	32	mtupmc1 Rx	mfupmc1 Rx mfpmc1 Rx
772	11000 00100	PMC2	no[6]	no[6]	32	mtupmc2 Rx	mfupmc2 Rx mfpmc2 Rx
773	11000 00101	PMC3	no[6]	no[6]	32	mtupmc3 Rx	mfupmc3 Rx mfpmc3 Rx
774	11000 00110	PMC4	no[6]	no[6]	32	mtupmc4 Rx	mfupmc4 Rx mfpmc4 Rx
775	11000 00111	PMC5	no[6]	no[6]	32	mtupmc5 Rx	mfupmc5 Rx mfpmc5 Rx
776	11000 01000	PMC6	no[6]	no[6]	32	mtupmc6 Rx	mfupmc6 Rx mfpmc6 Rx
779	11000 01011	MMCR0	no[6]	no[6]	64	mtummcr0 Rx	mfummcr0 Rx mfmmcr0 Rx

Table 5.1: SPR encodings (continued)

decimal	SPR[1]	Register Name	Privileged		Length (bits)	Extended Mnemonics*	
	$spr_{5:9}$ $spr_{0:4}$		mtspr	mfspr		mtspr	mfspr
780	11000 01100	SIAR	-	no[6]	64	-	mfusiar Rx mfsiar Rx
781	11000 01101	SDAR	-	no[6]	64	-	mfusdar Rx mfsdar Rx
782	11000 01110	MMCR1	-	no[6]	64	-	mfummcr1 Rx mfmmcr1 Rx
784	11000 10000	SIER	yes	yes	64	mtsier Rx	[11]
785	11000 10001	MMCR2	yes	yes	64	[11]	[11]
786	11000 10010	MMCRA	yes	yes	64	mtmmcra Rx	[11]
787	11000 10011	PMC1	yes	yes	32	mtpmc1 Rx	[11]
788	11000 10100	PMC2	yes	yes	32	mtpmc2 Rx	[11]
789	11000 10101	PMC3	yes	yes	32	mtpmc3 Rx	[11]
790	11000 10110	PMC4	yes	yes	32	mtpmc4 Rx	[11]
791	11000 10111	PMC5	yes	yes	32	mtpmc5 Rx	[11]
792	11000 11000	PMC6	yes	yes	32	mtpmc6 Rx	[11]
795	11000 11011	MMCR0	yes	yes	64	mtmmcr0 Rx	[11]
796	11000 11100	SIAR	yes	yes	64	mtsiar Rx	[11]
797	11000 11101	SDAR	yes	yes	64	mtsdar Rx	[11]
798	11000 11110	MMCR1	yes	yes	64	mtmmcr1 Rx	[11]
800	11001 00000	BESCRS[15]	no	no	64	mtbescrs Rx	mfbescrs Rx
801	11001 00001	BESCRSU[16]	no	no	32	mtbescrsu Rx	mfbescrsu Rx
802	11001 00010	BESCRR[15]	no	no	64	mtbescrr Rx	mfbescrr Rx
803	11001 00011	BESCRRU[16]	no	no	32	mtbescrru Rx	mfbescrru Rx
804	11001 00100	EBBHR	no	no	64	mtebbhr Rx	mfebbhr Rx
805	11001 00101	EBBRR	no	no	64	mtebbrr Rx	mfebbrr Rx
806	11001 00110	BESCR	no	no	64	mtbescr Rx	mfbescr Rx
808	11001 01000	reserved[8]	no	no	na	-	-
809	11001 01001	reserved[8]	no	no	na	-	-
810	11001 01010	reserved[8]	no	no	na	-	-
811	11001 01011	reserved[8]	no	no	na	-	-
812	11001 01100	DEXCR	-	no	32	-	mfudexcr Rx[14]
815	11001 01110	TAR	no	no	64	mttar Rx	mftar Rx
816	11001 10000	ASDR	hypv[2]	hypv[2]	64	mtasdr Rx	mfasdr Rx
823	11001 10111	PSSCR	yes	yes	64	mtpsscr Rx	mfpsscr Rx
828	11001 11100	DEXCR	yes	yes	64	mtdexcr Rx	mfdexcr Rx
848	11010 10000	IC	hypv[2]	yes	64	mtic Rx	mfic Rx
849	11010 10001	VTB	hypv[2]	yes	64	mtvtb Rx	mfvtb Rx
855	11010 10111	PSSCR	hypv[3]	hypv[3]	64	mthpsscr Rx	mfhpsscr
896	11100 00000	PPR	no	no	64	mtppr Rx	mfppr Rx
898	11100 00010	PPR32	no	no	32	mtppr32 Rx	mfppr32 Rx
1023	11111 11111	PIR	-	yes	32	-	mfpir Rx

Table 5.1: SPR encodings (continued)

decimal	SPR[1]	Register Name	Privileged		Length	Extended Mnemonics*	
	$spr_{5:9}$ $spr_{0:4}$		mtspr	mfspr	(bits)	mtspr	mfspr
- 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 AMOR before being placed into the UAMOR; see the ***mtspr*** instruction description. [6] $MMCR0_{PMCC}$ and $MMCR0_{PMCCEXT}$ controls the availability of this SPR, and its contents depend on the privilege state in which it is accessed. See Section 11.4.4 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] 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. [10] The ***mftb*** instruction is Phased-Out. Assemblers targeting Version 2.03 or later of the architecture should generate an ***mfspr*** instruction for the ***mftb*** and ***mftbu*** extended mnemonics; see the corresponding Assembler Note in the ***mftb*** instruction description (see Section 5.1 of Book II). [11] 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. [12] ***mtspr*** specifying this register is ultravisor privileged when $SMFCTRL_E$=1; otherwise it is hypervisor privileged. [13] This register is ultravisor privileged when $SMFCTRL_D$=1; otherwise it is hypervisor privileged. [14] “udexcr” in the extended mnemonic refers to the non-privileged SPR number for DEXCR. It is not to be confused with “UDEXCR”, which is the ultravisor equivalent of HDEXCR and cannot be read using ***mfspr*** instruction. See Section 9.3.3 of Book III. [15] This “register” is not an actual register, but rather an alias for the BESCR. When mtspr specifies the alias, the contents of register RS are written to the BESCR in a non-standard way, as specified in the mtspr instruction description. [16] This “register” is not an actual register, but rather an alias for the BESCRU. When mtspr specifies the alias, the contents of the low-order 32 bits of register RS are written to the BESCRU in a non-standard way, as specified in the mtspr instruction description.							
*This table 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

31	RS	spr	467	/
0	6	11	21	31

```
n ← spr5:9 || spr0:4
switch (n)
  case (13):
    if MSRHV PR = 0b10 then
      SPR(13) ← (RS)
    else if MSRHV PR = 0b00 then
      SPR(13) ← ((RS) & AMOR) | ((SPR(13)) & ¬AMOR)
    else
      SPR(13) ← ((RS) & UAMOR) | ((SPR(13)) & ¬UAMOR)
  case (29,61):
    if MSRHV PR = 0b10 then
      SPR(n) ← (RS)
    else
      SPR(n) ← ((RS) & AMOR) | ((SPR(n)) & ¬AMOR)
  case (157):
    if MSRHV PR = 0b10 then
      SPR(157) ← (RS)
    else
      SPR(157) ← (RS) & AMOR
  case (336):
    SPR(336) ← (SPR(336)) & (RS)
  case (800):
    BESCR ← BESCR | (RS)
  case (801):
    BESCRU ← BESCRU | (RS)32:63
  case (802):
    BESCR ← BESCR & ¬(RS)
  case (803):
    BESCRU ← BESCRU | ¬((RS)32:63)
  case (158, 808, 809, 810, 811):
  default:
    if length(SPR(n)) = 64 then
      SPR(n) ← (RS)
    else
      SPR(n) ← (RS)32:63
```

The SPR field denotes a Special Purpose Register, encoded as shown in Table 5.1. If the SPR field contains the value 158, the instruction is treated as a privileged no-op. If the SPR field contains a value from 808 through 811, the instruction specifies a reserved SPR, and is treated as a no-op; see Section 1.3.3, “Reserved Fields, Reserved Values, and Reserved SPRs” in Book I. Otherwise, except as described in the next six paragraphs, the contents of register RS, or of the low-order 32 bits of register RS if the designated Special Purpose Register is 32 bits long, are placed into the designated Special Purpose Register.

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 $MSR_{HV\ PR}$=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 MSR_{PR}=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 $MSR_{HV\ PR}$=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 Branch Event Status and Control Register (BESCR) using SPR 800 (BESCRS) or 802 (BESCRR), BESCR bits corresponding to 1 bits in register RS are modified and the remaining BESCR bits are not; for SPR 800 the BESCR bits corresponding to 1 bits in register RS are set to 1, and for SPR 802 the BESCR bits corresponding to 1 bits in register RS are set to 0. When the designated SPR is the Branch Event Status and Control Register Upper (BESCRU) using SPR 801 (BESCRSU) or 803 (BESCRRU), BESCRU bits corresponding to 1 bits in the low-order 32 bits of register RS are modified and the remaining BESCRU bits are not; for SPR 801 the BESCRU bits corresponding to 1 bits in the low-order 32 bits of register RS are set to 1, and for SPR 803 the BESCRU bits corresponding to 1 bits in the low-order 32 bits of register RS are set to 0.

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.

spr_0=1 if and only if writing the register is privileged. Execution of this instruction specifying an SPR number with spr_0=1 when the privilege state of the thread does not permit the access causes one of the following.

- MSR_{PR}=1: Privileged Instruction type Program interrupt
- $MSR_{HV\ PR}$=0b00 or $MSR_{S\ HV\ PR}$=0b010 and the SPR is always an ultravisor resource (independent of the contents of SMFCTRL): Privileged Instruction type Program interrupt
- $MSR_{HV\ PR}$=0b00 and the SPR is a hypervisor resource (see Table 5.1) or is PTCR, DAWRn, DAWRXn, or CIABR when they are ultravisor privileged for the operation:
  - $LPCR_{EVIRT}$=0: Privileged Instruction type Program interrupt
  - $LPCR_{EVIRT}$=1: Hypervisor Emulation Assistance interrupt
- $MSR_{S\ HV\ PR}$=0b010 and the SPR is PTCR, DAWRn, DAWRXn, or CIABR when they are ultravisor privileged for the operation: Hypervisor Emulation Assistance interrupt

Execution of this instruction specifying an SPR number that is undefined for the implementation causes one of the following.

- if spr_0=0:
  - if MSR_{PR}=1: Hypervisor Emulation Assistance interrupt
  - if MSR_{PR}=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 $LPCR_{EVIRT}$=0 and Hypervisor Emulation Assistance interrupt when $LPCR_{EVIRT}$=1 for all other SPRs
- if spr_0=1:
  - if MSR_{PR}=1: Privileged Instruction type Program interrupt
  - if MSR_{PR}=0: no operation (i.e., the instruction is treated as a no-op) when $LPCR_{EVIRT}$=0 and Hypervisor Emulation Assistance interrupt when $LPCR_{EVIRT}$=1

Special Registers Altered:

See Table 5.1

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. This maintains compatibility with POWER SPR encodings, in which these two instructions have only a 5-bit SPR field occupying bits 11:15.

Programming Note

For a discussion of software synchronization requirements when altering certain Special Purpose Registers, see 13 “Synchronization Requirements for Context Alterations” on page 1278.

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***/***mfspr*** specifying the corresponding SPRs as defined in the POWER Architecture.

Requiring that an attempt to execute an ***mtspr*** instruction with SPR=4, 5, or 6 cause a Hypervisor Emulation Assistance interrupt, even in privileged state, makes the behavior be the same for both instructions for all four SPR numbers, thereby simplifying the architecture. (SPRs 4, 5, and 6 were not defined for ***mtspr*** in the POWER Architecture. The corresponding SPRs were privileged for writing, and ***mtspr*** to those SPRs used the corresponding privileged SPR number.)

Move From Special Purpose Register XFX-form

mfspr RT,SPR

31	RT	spr	339	/
0	6	11	21	31

```
n ← spr5:9 || spr0:4
switch(n)
  case(800, 802): RT ← BESCR
  case(801, 803): RT ← 320 || BESCRU
  case(808, 809, 810, 811):
  default:
    if length(SPR(n)) = 64 then
      RT ← SPR(n)
    else
      RT ← 320 || SPR(n)
```

The SPR field denotes a Special Purpose Register, encoded as shown in Table 5.1. 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.

If the SPR field contains the value 800 (BESCRS) or 802 (BESCRR), the designated SPR is the BESCR. If the SPR field contains the value 801 (BESCRSU) or 803 (BESCRRU), the designated SPR is the BESCRU.

spr_0=1 if and only if reading the register is privileged. Execution of this instruction specifying an SPR number with spr_0=1 when the privilege state of the thread does not permit the access causes one of the following.

- MSR_{PR}=1: Privileged Instruction type Program interrupt
- $MSR_{HV\ PR}$=0b00 or $MSR_{S\ HV\ PR}$=0b010 and the SPR is always an ultravisor resource (independent of the contents of SMFCTRL): Privileged Instruction type Program interrupt
- $MSR_{HV\ PR}$=0b00 and the SPR is a hypervisor resource (see Table 5.1) or is DAWRn, DAWRXn, or CIABR when they are ultravisor privileged for the operation:
  - $LPCR_{EVIRT}$=0: Privileged Instruction type Program interrupt
  - $LPCR_{EVIRT}$=1: Hypervisor Emulation Assistance interrupt
- $MSR_{S\ HV\ PR}$=0b010 and the SPR is DAWRn, DAWRXn, or CIABR when they are ultravisor privileged for the operation: Hypervisor Emulation Assistance interrupt

Execution of this instruction specifying an SPR number that is not defined for the implementation causes one of the following.

- if spr_0=0:
  - if MSR_{PR}=1: Hypervisor Emulation Assistance interrupt
  - if MSR_{PR}=0: Hypervisor Emulation Assistance interrupt for SPRs 0, 4, 5, and 6, and no operation (i.e., the instruction is treated as a no-op) when $LPCR_{EVIRT}$=0 and Hypervisor Emulation Assistance interrupt when $LPCR_{EVIRT}$=1 for all other SPRs
- if spr_0=1:
  - if MSR_{PR}=1: Privileged Instruction type Program interrupt
  - if MSR_{PR}=0: no operation (i.e., the instruction is treated as a no-op) when $LPCR_{EVIRT}$=0 and Hypervisor Emulation Assistance interrupt when $LPCR_{EVIRT}$=1

Special Registers Altered:

None

Note

See the Notes that appear with ***mtspr***.

Move To Machine State Register X-form

mtmsr RS,L

31	RS	///	L	///	146	/
0	6	11	15	16	21	31

```
if L = 0 then
   MSR48 ← (RS)48 | (RS)49
   MSR58 ← ((RS)58 | (RS)49)
      & ¬(MSR41 & MSR3 & (¬(RS)49))
   MSR59 ← ((RS)59 | (RS)49)
      & ¬(MSR41 & MSR3 & (¬(RS)49))
   MSR32:40 42:47 49:50 52:57 60:62
      ← (RS)32:40 42:47 49:50 52:57 60:62
else
   MSR48 62 ← (RS)48 62
```

The MSR is set based on the contents of register RS and of the L field.

L=0:

The result of ORing bits 48 and 49 of register RS is placed into MSR_{48}. The result of ANDing bit 41 of the MSR with bit 3 of the MSR and with the complement of bit 49 of register RS is complemented and then ANDed with the result of ORing bits 58 and 49 of register RS and placed into MSR_{58}. The result of ANDing bit 41 of the MSR with bit 3 of the MSR and with the complement of bit 49 of register RS is complemented and then ANDed with the result of ORing bits 59 and 49 of register RS and placed into MSR_{59}. Bits 32:40, 42:47, 49:50, 52:57, and 60:62 of register RS are placed into the corresponding bits of the MSR.

L=1:

Bits 48 and 62 of register RS are placed into the corresponding bits of the MSR. The remaining bits of the MSR are unchanged.

This instruction is privileged.

If L=0 this instruction is context synchronizing. If L=1 this instruction is execution synchronizing; in addition, the alterations of the EE and RI bits take effect as soon as the instruction completes.

Special Registers Altered:

Register	*Field(s)*
MSR	

Extended Mnemonics:

Extended Mnemonics for *Move To Machine State Register*:

Extended mnemonic:	Equivalent to:
`mtmsr RS`	`mtmsr RS,0`

Except in the ***mtmsr*** instruction description in this section, references to "***mtmsr***" in this document imply either L value unless otherwise stated or obvious from context (e.g., a reference to an ***mtmsr*** instruction that modifies an MSR bit other than the EE or RI bit implies L=0).

Programming Note

If this instruction sets MSR_{PR} to 1, it also sets MSR_{EE}, MSR_{IR}, and MSR_{DR} to 1. If this instruction results in $MSR_{S\ HV\ PR}$ being equal to 0b110, it also sets MSR_{IR} and MSR_{DR} to 0.

This instruction does not alter MSR_{S}, 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.

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 7.9, "Interrupt Priorities" on page 1235). 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 13.

Programming Note

mtmsr serves as both a basic and an extended mnemonic. The Assembler will recognize an ***mtmsr*** mnemonic with two operands as the basic form, and an ***mtmsr*** mnemonic with one operand as the extended form. In the extended form the L operand is omitted and assumed to be 0.

Programming Note

There is no need for an analogous version of the ***mfmsr*** instruction, because the existing instruction copies the entire contents of the MSR to the selected GPR.

Move To Machine State Register Doubleword X-form

mtmsrd RS,L

31	RS	///	L	///	178	/
0	6	11	15	16	21	31

```
if L = 0 then
   MSR48 ← (RS)48 | (RS)49
   MSR58 ← ((RS)58 | (RS)49)
      & ¬(MSR41 & MSR3 & (¬(RS)49))
   MSR59 ← ((RS)59 | (RS)49)
      & ¬(MSR41 & MSR3 & (¬(RS)49))
   MSR0:2 4:40 42:47 49:50 52:57 60:62
      ← (RS)0:2 4 6:40 42:47 49:50 52:57 60:62
else
   MSR48 62 ← (RS)48 62
```

The MSR is set based on the contents of register RS and of the L field.

L=0:

The result of ORing bits 48 and 49 of register RS is placed into MSR_{48}. The result of ANDing bit 41 of the MSR with bit 3 of the MSR and with the complement of bit 49 of register RS is complemented and then ANDed with the result of ORing bits 58 and 49 of register RS and placed into MSR_{58}. The result of ANDing bit 41 of the MSR with bit 3 of the MSR and with the complement of bit 49 of register RS is complemented and then ANDed with the result of ORing bits 59 and 49 of register RS and placed into MSR_{59}. Bits 0:2, 4:40 42:47, 49:50, 52:57, and 60:62 of register RS are placed into the corresponding bits of the MSR.

L=1:

Bits 48 and 62 of register RS are placed into the corresponding bits of the MSR. The remaining bits of the MSR are unchanged.

This instruction is privileged.

If L=0 this instruction is context synchronizing. If L=1 this instruction is execution synchronizing; in addition, the alterations of the EE and RI bits take effect as soon as the instruction completes.

Special Registers Altered:

Register	*Field(s)*
MSR	

Extended Mnemonics:

Extended Mnemonics for *Move To Machine State Register Doubleword*:

Extended mnemonic:	Equivalent to:
`mtmsrd RS`	`mtmsrd RS,0`

Except in the ***mtmsrd*** instruction description in this section, references to "***mtmsrd***" in this document imply either L value unless otherwise stated or obvious from context (e.g., a reference to an ***mtmsrd*** instruction that modifies an MSR bit other than the EE or RI bit implies L=0).

Programming Note

If this instruction sets MSR_{PR} to 1, it also sets MSR_{EE}, MSR_{IR}, and MSR_{DR} to 1. If this instruction results in $MSR_{S\ HV\ PR}$ being equal to 0b110, it also sets MSR_{IR} and MSR_{DR} to 0.

This instruction does not alter MSR_{HV}, MSR_{S}, MSR_{ME}, or MSR_{LE}.

If the only MSR bits to be altered are $MSR_{EE\ RI}$, 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 7.9, "Interrupt Priorities" on page 1235). 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.

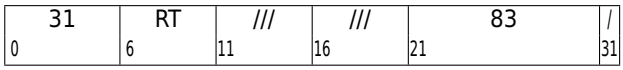
For a discussion of software synchronization requirements when altering certain MSR bits, see Chapter 13.

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 6. Storage Control

6.1 Overview

A program references storage using the effective address computed by the hardware when it executes a *Load*, *Store*, *Branch*, or *Cache Management* instruction, or when it fetches the next sequential instruction. The effective address is translated to a real address according to procedures described in Section 6.7.3, in Section 6.7.7 and in the following sections. The real address is what is presented to the storage subsystem.

For a complete discussion of storage addressing and effective address calculation, see Section 1.10 of Book I.

6.2 Storage Exceptions

A *storage exception* results when the sequential execution model requires that a storage access be performed but the access is not permitted (e.g., is not permitted by the storage protection mechanism), the access cannot be performed because the effective address cannot be translated to a real address, or the access matches some tracking mechanism criteria (e.g., Data Address Watchpoint).

In certain cases a storage exception may result in the "restart" of (re-execution of at least part of) a Load or Store instruction. See Section 2 of Book II, and Section 7.6 in this Book.

6.3 Instruction Fetch

Instructions are fetched under control of MSR_{IR}.

MSR_{IR}=0
: The effective address of the instruction is interpreted as described in Section 6.7.3.

MSR_{IR}=1
: The effective address of the instruction is translated by the Address Translation mechanism described beginning in Section 6.7.7.

6.3.1 Implicit Branch

Explicitly altering certain MSR bits (using ***mtmsr***[***d***]), or explicitly altering SLB entries, Page Table Entries, or certain System Registers (including the HRMOR, URMOR, and possibly other implementation-dependent registers), may have the side effect of changing the addresses, effective or real, from which the current instruction stream is being fetched. This side effect is called an *implicit branch*. For example, an ***mtmsrd*** instruction that changes the value of MSR_{SF} may change the effective addresses from which the current instruction stream is being fetched. The MSR bits and System Registers (excluding implementation-dependent registers) for which alteration can cause an implicit branch are indicated as such in 13 "Synchronization Requirements for Context Alterations" on page 1278. Implicit branches are not supported by the Power ISA. If an implicit branch occurs, the results are boundedly undefined.

6.3.2 Address Wrapping Combined with Changing MSR Bit SF

If the current instruction is at effective address $2^{32} - 4$ and is an ***mtmsrd*** instruction that changes the contents of MSR_{SF}, the effective address of the next sequential instruction is undefined.

Programming Note

If the thread is in 32-bit mode, the current instruction is a word instruction at effective address $2^{32} - 4$ or a prefixed instruction at effective address $2^{32} - 8$, and an interrupt occurs that is defined to set SRR0 or HSRR0 (or LR, for the System Call Vectored interrupt) to the effective address of the next sequential instruction, the contents of SRR0 or HSRR0 (or LR), as appropriate to the interrupt, are undefined.

6.4 Data Access

Data accesses are controlled by MSR_{DR}.

MSR_{DR}=0
: The effective address of the data is interpreted as described in Section 6.7.3.

MSR_{DR}=1
: The effective address of the data is translated by the Address Translation mechanism described in Section 6.7.7.

6.5 Performing Operations Out-of-Order

An operation is said to be performed "in-order" if, at the time that it is performed, it is known to be required by the sequential execution model. An operation is said to be performed "out-of-order" if, at the time that it is performed, it is not known to be required by the sequential execution model.

Operations are performed out-of-order on the expectation that the results will be needed by an instruction that will be required by the sequential execution model. Whether the results are really needed is contingent on everything that might divert the control flow away from the instruction, such as *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

  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 6.8.1.1.
- Executing instructions subsequent to an *ori R31,R31,0* instruction

  The restrictions for this case are given in Section 9.2.1.

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 6.7.12.
- 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.

Engineering Note

Out-of-order execution of the ***stbcx.***, ***sthcx.***, ***stwcx.***, ***stdcx.***, and ***stqcx.*** instructions is extremely complex and is not recommended.

Engineering Note

Because an External, Decrementer, or Performance Monitor exception can become pending at any time, it might seem that if MSR_{EE}=1 then fetching or executing any instruction beyond the current instruction is an out-of-order operation. However, these operations need not be treated as out-of-order if the taking of the interrupt is delayed until after they have completed. Similar considerations apply to Floating-Point Enabled Exception type Program interrupts when one of the Imprecise floating-point exception modes is in effect.

Engineering Note

Implementations that perform operations out-of-order must take care to obey the sequential execution model except as permitted by the architecture. Examples of cases that may require special attention include the following.

- Changes of control flow, including *System Call, Trap*, *Return From Interrupt,* and interrupts as well as branches.
- Changes of context due to changes of control flow. For example, the code at a branch target location, or the handler for System Call or Trap interrupts, may change the context and then return, so that the instructions immediately following the *Branch*, *System Call*, or *Trap* execute in a new context.
- Changes to resources that affect address translation, storage protection, or storage control attributes, when the change is followed by the appropriate software synchronization. Such resources include $MSR_{SF\ PR\ IR\ DR}$, PTCR, the Page Table, the SLB, and the TLB.
- Execution synchronizing and context synchronizing operations.

6.6 Invalid Real Address

A storage access (including an access that is performed out-of-order; see Section 6.5) may cause a Machine Check if the accessed storage location contains an uncorrectable error or does not exist.

In the case that the accessed storage location does not exist, the Checkstop state may be entered. See Section 7.5.2 on page 1206.

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.

6.7 Storage Addressing

Storage Control Overview

- Host real address space size is 2^m bytes, m≤60; see Note 1.
- Guest real address space size is 2^m bytes, m≤60; see Notes 1 and 2.
- Real page size is 2^{12} bytes (4 KB).
- Effective address space size is 2^{64} bytes.
- For HPT translation, an effective address is translated to a virtual address via a segment descriptor that was either bolted into the Segment Lookaside Buffer (SLB) by software or found and installed into the SLB via a hardware walk of the Segment Table. After that, the virtual address is translated to a host real address via a hardware walk of the Page Table.
  - Virtual address space size is 2^n bytes, 65≤n≤78; see Note 3.
  - Segment size is 2^s bytes, s=28 or 40.
  - $2^{n-40} \leq$ number of virtual segments $\leq 2^{n-28}$; see Note 3.
  - Virtual page size is 2^p bytes, where 12≤p, and 2^p is no larger than either the size of the biggest segment or the real address space; a size of 4 KB, 64 KB, and an implementation-dependent number of other sizes are supported; see Note 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≤p, and 2^p is no larger than the size of the real address space; a size of 4 KB, 64 KB, 2MB, and an implementation-dependent number of other sizes are supported; see Note 4. The 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.

6.7.1 32-Bit Mode

The computation of the 64-bit effective address is independent of whether the thread is in 32-bit mode or 64-bit mode. In 32-bit mode (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.

6.7.2 Virtualized Partition Memory (VPM) Mode

VPM mode enables the hypervisor to reassign all or part of a partition's memory transparently so that the reassignment is not visible to the partition. When this is done, the partition's memory is said to be "virtualized." This mode is only available within Paravirtualized HPT translation mode. Radix Tree translation mode provides equivalent

function by providing two levels of translation with separate Page Tables for the operating system and the hypervisor. (See Section 6.7.7 for a more complete overview of the translation modes.) The VPM field in the LPCR enables VPM mode when address translation is enabled. VPM is always enabled when address translation is disabled.

If the thread is not in hypervisor state, and either address translation is enabled and VPM=1, or address translation is disabled, conditions that would have caused a Data Storage or an Instruction Storage interrupt if the affected memory were not virtualized instead cause a Hypervisor Data Storage or a Hypervisor Instruction Storage interrupt respectively. Because the Hypervisor Data Storage and Hypervisor Instruction Storage interrupts always put the thread in hypervisor state, they permit the hypervisor to handle the condition if appropriate (e.g., to restore the contents of a page that was reassigned), and to reflect it to the operating system's Data Storage or Instruction Storage interrupt handler otherwise.

When address translation is enabled, VPM mode has no effect on address translation. When address translation is disabled, addressing is controlled as specified in Section 6.7.3.

6.7.3 Ultravisor Real, Hypervisor Real, and Virtual Real Addressing Modes

If a storage access is an instruction fetch performed when instruction address translation is disabled, or if the access is a data access performed when data address translation is disabled, it is said to be performed in "ultravisor real addressing mode" if the thread is in ultravisor state, in "hypervisor real addressing mode" if the thread is in hypervisor non-ultravisor state, and in "virtual real addressing mode" if the thread is in privileged non-hypervisor state. Storage accesses in ultravisor real, hypervisor real, and virtual real addressing modes are performed in a manner that depends on the contents of $MSR_{S\,HV}$, $PATE_{HR}$, $PATE_{PS}$, URMOR (see Chapter 3), HRMOR (see Chapter 2), bit 0 of the effective address (EA_0), and the state of the Real Mode Storage Control Facility as described below. Bits 1:3 of the effective address are ignored.

$MSR_{S\,HV}$=0b11

- If EA_0=0, the Ultravisor Offset Real Mode Address mechanism, described in Section 6.7.3.1, controls the access.
- If EA_0=1, bits 4:63 of the effective address are used as the real address for the access.

$MSR_{S\,HV}$=0b01

- If EA_0=0, the Hypervisor Offset Real Mode Address mechanism, described in Section 6.7.3.1, controls the access.
- If EA_0=1, bits 4:63 of the effective address are used as the real address for the access.

MSR_{HV}=0

- If $PATE_{HR}$=0, the Virtual Real Mode Addressing mechanism, described in Section 6.7.3.3, controls the access.
- If $PATE_{HR}$=1, partition-scoped translation is performed on the effective address. (See Section 6.7.11.3, "Obtaining Host Real Address, Radix on Radix".)

Engineering Note

Because EA_0 is the last effective address bit to become available as a result of effective address computation, using EA_0 to determine how to compute the real address when MSR_{HV} = 1, as described above, has the potential to degrade performance. One technique for avoiding such degradation is to use the ERAT to "translate" all effective addresses to real addresses (i.e., to use the ERAT even when address translation is disabled).

If the preceding technique is implemented, care must be taken to ensure that a given ERAT entry is used to "translate" a given effective address only if the "translation" mechanism that was used when the entry was created is the same as the mechanism that would be used if no ERAT entry existed.

6.7.3.1 Ultravisor/Hypervisor Offset Real Mode Address

If MSR_{HV} = 1 and EA_0 = 0, the access is controlled by the contents of the Ultravisor Real Mode Offset Register or the Hypervisor Real Mode Offset Register, depending on the value of MSR_S, as follows.

Ultravisor Real Mode Offset Register (URMOR)
When MSR_S=1, bits 4:63 of the effective address for the access are ORed with the 60-bit offset represented by the contents of the URMOR, and the 60-bit result is used as the real address for the access.

Hypervisor Real Mode Offset Register (HRMOR)
When MSR_S=0, bits 4:63 of the effective address for the access are ORed with the 60-bit offset represented by the contents of the HRMOR, and the 60-bit result is used as the real address for the access.

For each of the two registers, the supported offset values are all values of the form $i \times 2^r$, 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

$EA_{4:63-r}$ should equal $^{60-r}0$. If this condition is satisfied, ORing the effective address with the offset produces a result that is equivalent to adding the effective address and the offset.

If m<60, $EA_{4:63-m}$, $URMOR_{4:63-m}$, and $HRMOR_{4:63-m}$ must be zeros.

Engineering Note

The low-order r bits of the URMOR and the HRMOR need not be implemented. If m<60, bits 4:63-m of the URMOR and the HRMOR need not be implemented. Bits that are not implemented must be treated as if they were zeros.

6.7.3.2 Storage Control Attributes for Accesses in Ultravisor and Hypervisor Real Addressing Modes

Storage accesses in ultravisor and hypervisor real addressing modes are performed as though all of storage had the following storage control attributes, except as modified by the Hypervisor Real Mode Storage Control facility (see Section 6.7.3.2.1). (The storage control attributes are defined in Book II.)

- not Write Through Required
- not Caching Inhibited, for instruction fetches
- not Caching Inhibited, for data accesses except those caused by the *Load/Store Caching Inhibited* instructions; Caching Inhibited, for data accesses caused by the *Load/Store Caching Inhibited* instructions
- Memory Coherence Required, for data accesses
- Guarded
- []

Additionally, storage accesses in ultravisor and hypervisor real addressing modes are performed as though all storage was not No-execute.

Programming Note

Because storage accesses in ultravisor and hypervisor real addressing modes do not use the SLB or the Page Table, accesses in these modes bypass all checking and recording of information contained therein (e.g., storage protection checks that use information contained therein are not performed, and reference and change information is not recorded).

6.7.3.2.1 Hypervisor Real Mode Storage Control

The Hypervisor Real Mode Storage Control facility provides a means of specifying portions of real storage that are treated as neither Caching Inhibited nor Guarded in ultravisor and hypervisor real addressing modes ($MSR_{HV\ PR}$=0b10, and MSR_{IR}=0 or MSR_{DR}=0, as appropriate for the type of access). The remaining portions are treated as Caching Inhibited and Guarded in ultravisor and hypervisor real addressing modes. The means is a hypervisor resource (see Chapter 2), and may also be system-specific.

The facility divides real storage into history blocks, in implementation-specific sizes. The history for instruction fetches is tracked separately from that for data accesses. If there is no instruction fetch history for a block and it is the target of an instruction fetch, the access is performed as though the block is Guarded, but the block is treated as not Guarded for subsequent instruction fetches on a best effort basis, limited by the amount of history that the facility can maintain. If there is no data access history for a block and it is accessed using a *Load/Store Caching Inhibited* instruction, the access is performed as though the block is Guarded, and the block is treated as Guarded for subsequent accesses on a best effort basis, limited by the amount of history that the facility can maintain. If there is no data access history for a block and it is accessed using any other *Load* or *Store* instruction, the access is performed as though the block is Guarded, but the block is treated as not Guarded for subsequent accesses on a best effort basis, limited by the amount of history that the facility can maintain. If the history causes a block to be treated as Guarded, the block is also treated as Caching Inhibited; if the history causes a block to be treated as not Guarded, the block is also treated as not Caching Inhibited.

If the storage location specified by a *Load/Store Caching Inhibited* instruction is in storage that is specified by the Hypervisor Real Mode Storage Control facility to be treated as not Guarded, a Data Storage interrupt occurs. ("specified by the Hypervisor Real Mode Storage Control facility" means "specified in a history block".) The history can be erased using an slbia instruction; see Section 6.9.3.2.

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 is that if a *Load* or *Store* instruction specifies a location in a block for which history exists and was established by the other type of *Load*/*Store*, the data access will perform less well than it otherwise would (another type of *Load*/*Store* and history was established by *Load*/*Store Caching Inhibited*) or will cause a Data Storage interrupt (*Load*/*Store Caching Inhibited* and history was established by another type of *Load*/*Store*). The granularity for concern is the history block. For this caution, instruction fetches are irrelevant because they have their own history mechanism and are always intended to be treated as neither Caching Inhibited nor 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 6.8.2.2, "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 ultravisor 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 6.7.3.4.

Programming Note

The preceding capability can be used to improve the performance of software that runs in ultravisor and hypervisor real addressing modes, by causing accesses to instructions and data that occupy well-behaved storage to be treated as neither Caching Inhibited nor Guarded.

6.7.3.3 Virtual Real Mode Addressing Mechanism

If MSR_{HV}=0, the partition is using Paravirtualized HPT translation ($PATE_{HR}$=0), and MSR_{DR}=0 or MSR_{IR}=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 6.1 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.

Field	Value
ESID	$^{36}0$
V	1
B	0b01 – 1 TB
VSID	0b00 || 0x0_01FF_FFFF
K_s	0
K_p	undefined
N	0
L	$PATE_{PS[0]}$
C	0
LP	$PATE_{PS[1:2]}$

Figure 6.1: SLBE for VRMA

Programming Note

The C bit in Figure 6.1 is set to 0 because the implementation-specific lookaside information associated with the VRMA is expected to be long-lived. See the Programming Note about Class in Section 6.7.8.1.

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 6.1.

Architecture Note

The 1 TB VSID was chosen to have a fixed number of 0s and 1s (instead of 0s concatenated with 1s in all implemented VSID bits) so that the VSID does not change after partition migration to a thread with a different number of implemented VSID bits.

6.7.3.4 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
- []

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.

6.7.4 Definitions

translation mode: Refers to either HPT translation or Radix Tree translation. The translation mode is specified by the HR field in the Partition Table Entry corresponding to the contents of the LPIDR.

process-scoped: Refers to translation performed using tables pointed to by Process Table Entries: guest Radix Tree translation, host Radix Tree translation for quadrants 0 and 3 when 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 "eff" is used to indicate the possibility of such a substitution. This value substitution happens only in Radix Tree translation, and is based on the value of $EA_{0:1}$ (see Section 6.7.5.1, "Effective Address Space Structure for Radix-using Partitions"). Value substitution does not happen in HPT translation. When a guest uses Radix Tree translation, PID substitution may take place. When a host uses Radix Tree translation, both PID and LPID substitution may take place. When a host uses HPT translation, the only special significance associated with LPIDR=0 is with regard to Segment Table walk when MSR_{HV}=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 $MSR_{S\ HV\ PR}$=0b011, thereby restricting the resources it can modify (MSR_{PR}=1) and causing its interrupts to go to the hypervisor ($MSR_{S\ HV}$=0b01). It shares an HPT with the partition it serves. The adjunct's storage is kept separate from the client partition's storage using Virtual Page Class Key protection. (The adjunct's lightness of weight derives from not requiring a full partition context switch (SLB flush, TLB flush, 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 an HR=0 hypervisor and only for LPID$\neq$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.

Architecture Note

If the architecture changes to a multi-context SLB in the future and segment descriptors are only invalidated to reuse LPIDs or PIDs, an adjunct bit will be necessary to differentiate between the two distinct EA spaces.

6.7.5 Address Ranges Having Defined Uses

The address ranges described below have uses that are defined by the architecture.

- Fixed interrupt vectors

  Except for the first 256 bytes, which are reserved for software use, the real page beginning at real address 0x0000_0000_0000_0000 is either used for interrupt vectors or reserved for future interrupt vectors.

- Implementation-specific use

  The two contiguous real pages beginning at real address 0x0000_0000_0000_1000 are reserved for implementation-specific purposes.

- Offset Real Mode interrupt vectors

  The real pages beginning at the real addresses specified by the URMOR and the HRMOR are used similarly to the page for the fixed interrupt vectors.

- Relocated interrupt vectors

  Depending on the values of $MSR_{S\ HV\ IR\ DR}$ when the interrupt occurs and on the value of $LPCR_{AIL}$ or $LPCR_{HAIL}$ as appropriate, the virtual page containing the byte addressed by effective address [] 0xC000_0000_0000_4000 may be used similarly to the page for the fixed interrupt vectors. (See Section 2.2.)

- *System Call Vectored* interrupt vectors

  Depending on the values of $MSR_{S\ HV\ IR\ DR}$ when the interrupt occurs and on the value of $LPCR_{AIL}$ or $LPCR_{HAIL}$ as appropriate, 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. (See Section 2.2.)

- Partition Table

  A contiguous sequence of real pages beginning at the real address specified by the PTCR contains the Partition Table.

- Page Table

  A contiguous sequence of real pages beginning at the real address specified by the first doubleword of the Partition Table Entry when HR=0 contains the Page Table.

6.7.5.1 Effective Address Space Structure for Radix-using Partitions

When Radix Tree translation is in use but translation is disabled (MSR_{IR}=0 or MSR_{DR}=0, as appropriate for the type of access), MSR_{HV} selects between partition-scoped translation of the real mode guest real address, formed by treating $EA_{0:1}$ as 0b00, and hypervisor or ultravisor real mode (see Section 6.7.3). When Radix Tree translation is in use and translation is enabled, $EA_{0:1}$ together with MSR_{HV} are used to select one of as many as three distinct Radix Trees 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 6.2 for an illustration of the mappings.) Since there's nothing to prevent a process from generating any address in the 64b EA space, the exceptional cases are defined as follows. When a quadrant of the EA space has no associated Radix Tree, access to it results in an Instruction Segment exception or Data Segment exception, as appropriate for the type of access. Similarly, reference to any portion of these quadrants or the real mode guest real address described above that is not mapped by a Radix Tree (versus mapped by an invalid entry) will cause an Instruction or Data Segment exception.

Programming Note

Note that the quadrant structure is only available to software running in 64b mode with address translation enabled. 32b software will only be able to access storage mapped by its own Radix Tree. When address translation is disabled and HV||PR=0b00, the EA accesses storage mapped into the guest real address space.

Programming Note

Warning: The functionality described in this section, e.g. directing most hypervisor interrupts to the 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, quadrant 0 ($EA_{0:1}$=0b00) is mapped by the Radix Tree for the application. For guest operating systems and the hypervisor acting as an operating system (LPIDR=0), quadrant 0 is mapped by the Radix Tree for the application and quadrant 3 ($EA_{0:1}$=0b11) is mapped by the Radix Tree for the direct supervisor of the application. Quadrants 1 and 2 have no associated Radix Tree for guest and host applications and guest operating systems, but hold echoes of quadrants 0 and 3 for the hypervisor acting as an operating system.

Programming Note

Outboard accelerators may commonly be limited to accessing quadrant 0 as a matter of platform architecture. In such platforms, references to other quadrants [] may be regarded as errors.

For the hypervisor acting as a hypervisor (LPIDR≠0), quadrant 3 is as described above. Quadrant 1 ($EA_{0:1}$=0b01) is mapped by the Radix Tree for the guest application and quadrant 2 ($EA_{0:1}$=0b10) is mapped by the Radix Tree for the guest operating system, one of which experienced a hypervisor interrupt or performed a system call to the hypervisor. Quadrant 0 has no associated Radix Tree.

When MSR_{HV}=1 and $EA_{0:1}$=0b00 or 0b11 (and the quadrant is mapped by a Radix Tree), only process-scoped translation is performed. When MSR_{HV}=0 and $MSR_{IR/DR}$=0, only partition-scoped translation is performed. Otherwise, nested process- and partition-scoped translations are performed.

Engineering Note

The hypervisor's quadrant 3 ($MSR_{HV\ PR}$=0b10 and $EA_{0:1}$=0b11) is the same Radix Tree for the entire system. The design may cache a single value for the base address of this tree, to be shared by all threads in the core.

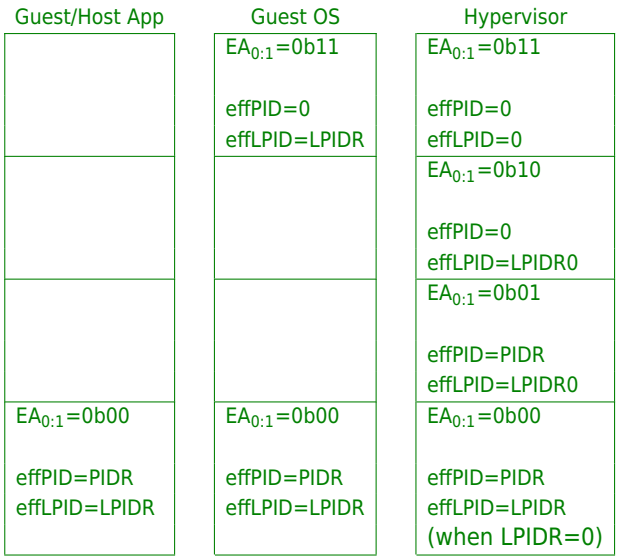


Figure 6.2: Effective address space structure when using Radix Tree translation

6.7.6 In-Memory Tables

The In-Memory Tables are used to find the tables that are used in the actual translation process for the partition and process that are executing. They enable hardware, including accelerator hardware separate and distinct from the Power ISA processors in the platform, to perform the translation process largely without software intervention. Description of the In-Memory Table structure follows. Hardware may cache the contents of the In-Memory Tables. Variants of ***tlbie***[***l***] may be used to manage the caching even though the In-Memory Table contents are not cached in the TLB. When "thread" is used in descriptions of the ordering of accesses and operations (e.g. invalidations) related to translation cache management, it should be understood to include execution streams in accelerators unless otherwise stated or obvious from context.

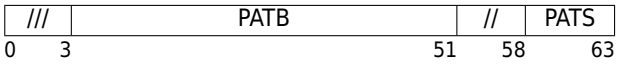
When an address in the In-Memory Table structure is specified to be a virtual or guest real address, the access to that address is considered to be performed with translation on. For a host using HPT translation, a base page size is specified for each such access to be used in the HPT search. The hypervisor can override the Segment Table Page Size in the Process Table Entry ($PRTE_{STPS,}$ see Figure 6.5) using $LPCR_{ISL}$. The base page size for the Process Table ($PATE_{PRTPS}$) 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 MSR_{PR}=0 for the purpose of determing storage protection, although instruction side translation exceptions cause [H]ISI. (A specific example of the implications of this is that tables used to translate instruction fetches may be located in guarded or no-execute storage.)

Programming Note

The descriptors in the entries in this section and its subsections contain addresses that are properly aligned so that no shifting is required. For example, the minimum size of the Partition Table is 4KB, so PATB has the thirteenth least significant address bit as its least significant bit. To construct the real address for a 4KB table, 12 zeros are appended on the right, and an appropriate number of address bits are removed from the left to match the real address size (m) supported by the implementation. For an aligned 8K table, bit 51 of the PTCR would be disregarded, and 13 zeros would be appended.

6.7.6.1 Partition Table

The Partition Table Control Register (PTCR) is a 64-bit register that contains the host real address of the base of the Partition Table and specifies its size. Software must ensure that the contents of the PTCR are the same for all processors in the system prior to enabling translation or transferring control to a partition.



Partition Descriptor

Bit(s)	Name	Description
4:51	PATB	Partition Table Base
59:63	PATS	Partition Table Size=$2^{12+PATS}$, PATS$\leq$24

All other fields are reserved.

Figure 6.3: Partition Table Control Register

Programming Note

If it becomes necessary to shrink the Partition Table or to change PATB to point to a table that is not identical to the existing one, it is necessary to issue ***tlbie*** with RIC=2 to invalidate caching of outdated In-Memory Table Entries.

The Partition Table is composed of a pair of doublewords per partition. The first doubleword indicates whether the partition uses HPT or Radix Tree translation and whether the partition is secure, and contains the base of the host's translation table structure in host real memory. The first doubleword also contains the size of the table structure and the size of the Root Page Directory for a hypervisor using Radix Tree translation, or the base page size for the VRMA for Paravirtualized HPT translation. Additional details about the parameters for HPT translation follow.

The HTABORG field contains the high-order 42 bits of the 60-bit real address of the Page Table. The Page Table is thus constrained to lie on a 2^{18} byte (256 KB) boundary. At least 11 bits from the hash function (see Figure 6.12) are used to index into the Page Table. The minimum size Page Table is 256 KB (2^{11} PTEGs of 128 bytes each).

The Page Table can be any size 2^n bytes where $18 \leq n \leq 46$. As the table size is increased, more bits are used from the hash to index into the table.

The HTABSIZE field contains an integer giving the number of bits (in addition to the minimum of 11 bits) from the hash that are used in the Page Table index. This number must not exceed 28. HTABSIZE is used to generate a mask of the form 0b00...011...1, which is a string of 28 - HTABSIZE 0-bits followed by a string of HTABSIZE 1-bits. The 1-bits determine which additional bits (beyond the minimum of 11) from the hash are used in the index (see Figure 6.12).

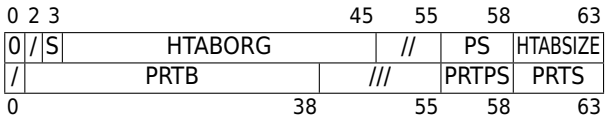
On implementations that support a real address size of only m bits, m$<$60, bits 0:59-m of the HTABORG field are treated as reserved bits, and software must set them to zeros.

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 6.7.9.2) 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.

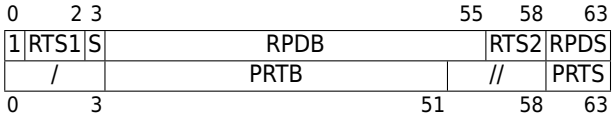
Example:

Suppose that the Page Table is 16,384 (2^{14}) 128-byte PTEGs, for a total size of 2^{21} bytes (2 MB). A 14-bit index is required. Eleven bits are provided from the hash to start with, so 3 additional bits from the hash must be selected. Thus the value in HTABSIZE must be 3. The HPT may begin on any 256KB boundary.



Paravirtualized HPT Partition Table Entry

Bit(s)	Name	Description
0	HR	Host Radix 0b0 - hypervisor uses HPT translation for this partition 0b1 - hypervisor uses Radix Tree translation for this partition
3	S	Partition is Secure
4:45	HTABORG	Hashed Page Table Base
56:58	PS	Page Size (uses L||LP encoding as in SLBE)
59:63	HTABSIZE	HPT size = $2^{HTABSIZE+18}$, HTABSIZE $\leq$ 28
1:38	PRTB	Process Table Base (when UPRT=1)
56:58	PRTPS	Process Table Page Size, uses L||LP encoding as in SLBE (when UPRT=1)
59:63	PRTS	Process Table Size = $2^{12+PRTS}$, PRTS$\leq$24 (when UPRT=1)



Radix on Radix Partition Table Entry

Bit(s)	Name	Description
0	HR	Host Radix 0b0- hypervisor uses HPT translation for this partition 0b1- hypervisor uses Radix Tree translation for this partition
1:2	RTS1	Radix Tree Size[0:1]
3	S	Partition is Secure
4:55	RPDB	Root Page Directory Base
56:58	RTS2	Radix Tree Size[2:4] (number of address bits mapped), size=2^{RTS+31}
59:63	RPDS	Root Page Directory Size = 2^{RPDS+3}, RPDS$\geq$5
4:51	PRTB	Process Table Base
59:63	PRTS	Process Table Size = $2^{12+PRTS}$, PRTS$\leq$24 (when UPRT=1)

All other fields are reserved.

Figure 6.4: 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 a radix partition and a virtual address for an HPT partition, 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. For an HPT partition, 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.

The second doubleword also includes the base page size used for the HPT search. Address translation and storage protection for the Process Table Entry are handled as if address translation were enabled using an implied segment descriptor with base page size as just described, B=0b01 (1TB segment), K_s=K_p=0, N=0, and C=0, and the Virtual Page Class Key Protection mechanism does not apply. The Partition Table Entry variants are illustrated in Figure 6.4. 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 MSR_{HV}=1. In addition, LPID=0 with Radix Tree translation is an unsupported MMU configuration when MSR_{HV}=0. These two unsupported MMU configurations, plus the case of UPRT being 0 when HR=1, constitute translation mode mismatches.

Programming Note

The S bit in Partition Table Entries is provided for use by outboard mechanisms that access storage. The processor uses MSR_S, not $PATE_S$, to determine partition security.

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. (An out-of-range PID will not produce an exception simply because of its size.) Hypervisor software may protect against such errors by the OS by not providing a translation for virtual / guest real addresses beyond the end of the Process Table.

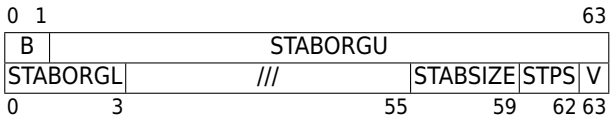
Similarly, no size checking is provided for the Partition Table. (An out-of-range LPID will not produce an exception simply because of its size.)

Engineering Note

It is highly desirable for hardware to generate the address for an out-of-bounds access to a Process Table so that software can detect its error by not providing a translation for addresses beyond the table.

6.7.6.2 Process Table

The Process Table is composed of a quadword Process Table Entry per process in the partition. For partitions that use HPT translation, the Process Table Entry contains a Segment Table descriptor, which is composed of the origin of the Segment Table in virtual address space, the segment size and base page size for the segment [] that holds 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. Address translation and storage protection for the Segment Table Entry Group are handled as if address translation were enabled using an implied segment descriptor with segment size and base page size as just described, K_s=K_p=0, N=0, and C=0, and the Virtual Page Class Key Protection mechanism 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).



DW	Bit(s)	Name	Description
0	0:1	B	Segment Table Segment Size
	2:63	STABORGU	Segment Table Origin Upper ($VA_{0:61}$)
1	0:3	STABORGL	Segment Table Origin Lower ($VA_{62:65}$)
	56:59	STABSIZE	Segment Table Size = $2^{12+STABSIZE}$, STABSIZE $\leq$ 12
	60:62	STPS	Segment Table Page Size (uses L||LP encoding as in SLBE)
	63	V	Valid

DW	Bit(s)	Name	Description
0	1:2	RTS1	Radix Tree Size[0:1]
	3	/	Reserved
	4:55	RPDB	Root Page Directory Base
	56:58	RTS2	Radix Tree Size[2:4] (number of address bits mapped), size=2^{RTS+31}
	59:63	RPDS	Root Page Directory Size = 2^{RPDS+3}, RPDS$\geq$5

All other fields are reserved.

Figure 6.5: Process Table Entry Variants

6.7.7 Address Translation Overview

The effective address (EA) is the address generated by the hardware for an instruction fetch or for a data access. If address translation is enabled, this address is passed to the Address Translation mechanism, which attempts to convert the address to a real address which is then used to access storage. If the effective address cannot be translated, a storage exception (see Section 6.2) occurs.

The architecture defines segment translation and two types of page translation. Segment translation is paired with HPT translation. The other supported "pairing" is two level Radix Tree translation. Either of these pairings can be used to translate an effective address into a host real address. The In-Memory Tables described above determine the translation mode used by a partition, as well as the locations of the Page Tables and Segment Tables, and the base page size for the Segment Tables. When MSR_{HV}=1 and/or MSR_{IR}=0 or MSR_{DR}=0 (as appropriate for the type of access), the steps taken for a given mode vary. See Sections 6.7.11.3 and 6.7.11.4 for details.

The pairing of Segment translation and Hashed Page Table (HPT) translation applies Segment translation to an effective address to produce a virtual address as described in Section 6.7.8, and HPT translation to the virtual address to produce a host real address as described in Section 6.7.9. Segment translations can be established by both the guest and the hypervisor, but the HPT translation is always managed by the hypervisor with the guest typically giving direction via system calls to the hypervisor in a paravirtualization relationship. This mode is commonly referred to as Paravirtualized HPT translation. The segment translation is managed on a per-process ("process-scoped") basis, mapping a smaller effective address space into a large "partition-scoped" virtual address space, where the segment can be used as a shared memory object. There is also the possibility of thread-unique mappings. In the basic version of HPT translation, storage exceptions are directed to the operating system, which in turn issues system calls to the hypervisor. When Virtualized Partition Memory is enabled, storage exceptions are directed to the hypervisor, enabling a higher degree of memory overcommitment as the hypervisor transparently steals pages from the partition. Figure 6.6 gives an overview of the address translation process.

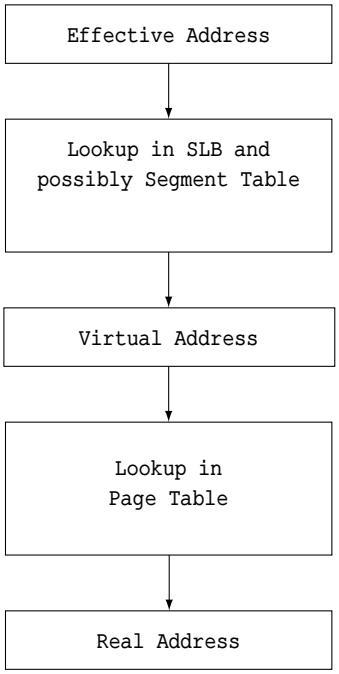


Figure 6.6: Address translation overview

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 6.7 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 6.18 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.

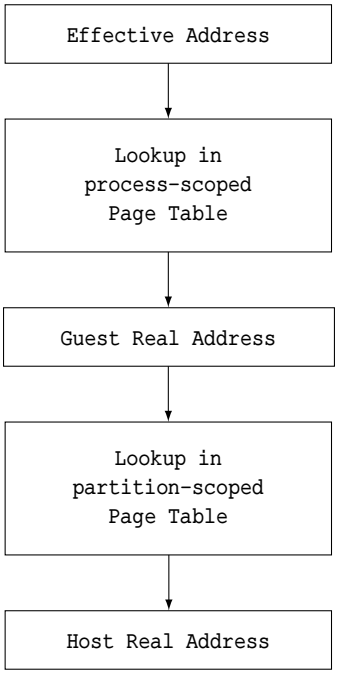


Figure 6.7: Address translation overview, Radix on Radix

Translation Lookaside Buffer

Conceptually, the Page Table is searched by the address relocation hardware to translate every reference. For performance reasons, the hardware usually keeps a Translation Lookaside Buffer (TLB) that holds PTEs that have recently been used. (Page-size-related information in TLB entries and its use in match checks may differ from that in PTEs; see Section 6.7.9.2 and Section 6.7.10.3.) The TLB is searched prior to searching the Page Table and, for Radix Tree Translation, prior to searching the Page Walk Cache. As a consequence, when software makes changes to the Page Table it must perform the appropriate TLB invalidate operations to maintain the consistency of the TLB with the Page Table (see Section 6.10). An implementation may associate each of its TLB entries with the partition for which the TLB entry was created, so that the entries can be retained while other partitions are executing.

Hardware generally will not create more than one TLB entry to translate a given 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 (HPT translation) or when software fails to invalidate the corresponding TLB entry when changing a page size. If a TLB search during address translation finds multiple matching TLB entries cre-

ated from such PTEs, one of the matching TLB entries is used as if it were the only matching entry, or a Machine Check occurs. Software should scrupulously avoid creating such mappings.

Programming Notes

1. Page Table Entries may or may not be cached in a TLB.
2. It is possible that the hardware implements more than one TLB, such as one for data and one for instructions. In this case the size and shape of the TLBs may differ, as may the values contained therein.
3. Use the ***tlbie*** instruction to ensure that the TLB no longer contains a mapping for a particular page.

Page Walk Cache

For performance reasons, the hardware usually keeps a Page Walk Cache (PWC) that holds Page Directory Entries that represent partial tree traversals (one or more levels) from recent Radix Tree translations. The PWC is searched (perhaps iteratively, depending on the design) with the goal of skipping some of the storage accesses that would otherwise be needed to traverse the Radix Tree. The internal structures of the Radix Trees are considered to be managed separately from the final translations. When software changes this structure, it must perform appropriate invalidations to the PWC to maintain the consistency of the PWC with the Radix Tree (see Section 6.10). An implementation may associate each of its PWC entries with the partition for which the PWC entry was created, so that the entries can be retained while other partitions are executing.

Programming Notes

1. Page Directory Entries may or may not be cached in a PWC.
2. It is possible that the hardware implements more than one PWC, such as one for data and one for instructions. In this case the size and shape of the PWCs may differ, as may the values contained therein.
3. Use the ***tlbie*** instruction to ensure that the PWC no longer contains information describing a particular portion of a Radix Tree.

6.7.8 Segment Translation

Conversion of a 64-bit effective address to a virtual address is done by searching the Segment Lookaside Buffer (SLB) as shown in Figure 6.8. If no matching translation is found in the SLB, $LPCR_{UPRT}$=1, and either MSR_{HV}=0 or LPID=0, the Segment Table is searched. For implicit accesses, implicit segment descriptors are provided, as described elsewhere in this chapter.

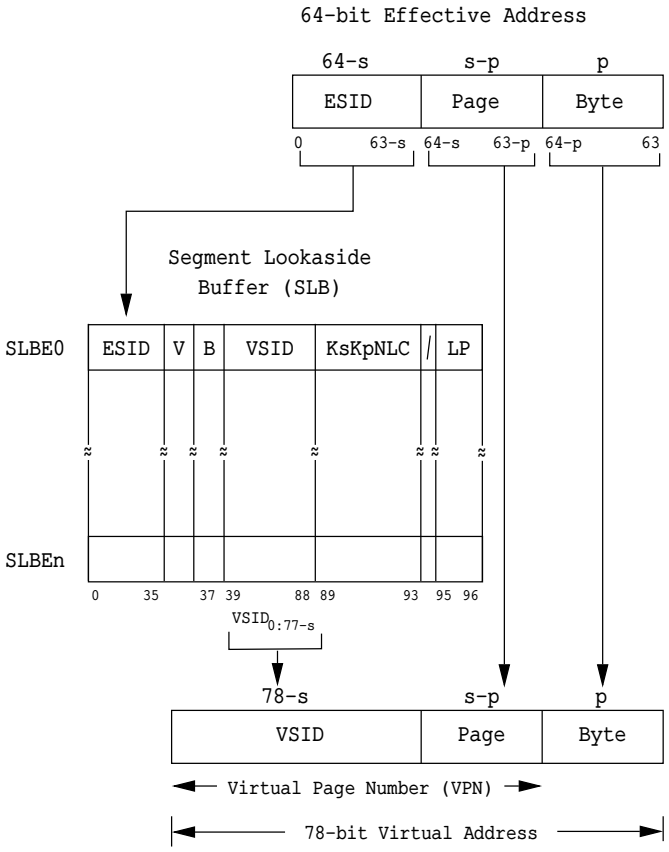


Figure 6.8: Translation of 64-bit effective address to 78 bit virtual address

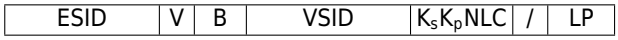
6.7.8.1 Segment Lookaside Buffer (SLB)

The Segment Lookaside Buffer (SLB) specifies the mapping between Effective Segment IDs (ESIDs) and Virtual Segment IDs (VSIDs). The number of SLB entries is implementation-dependent, except that all implementations provide at least 32 entries.

The first four entries, and when $LPCR_{UPRT}$=0 all of the entries, of the SLB are managed by software, using the instructions described in Section 6.9.3.2. See 13 “Synchronization Requirements for Context Alterations” on page 1278 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 6.9 shows the layout of an SLB entry.



Bit(s)	Name	Description
0:35	ESID	Effective Segment ID
36	V	Entry valid (V=1) or invalid (V=0)
37:38	B	Segment Size Selector 0b00 - 256 MB (s=28) 0b01 - 1 TB (s=40) 0b10 - reserved 0b11 - reserved
39:88	VSID	Virtual Segment ID
89	K_s	Supervisor (privileged) state storage key (see Section 6.7.13.2)
90	K_p	Problem state storage key (See Section 6.7.13.2.)
91	N	No-execute segment if N=1
92	L	Virtual page size selector bit 0
93	C	Class
95:96	LP	Virtual page size selector bits 1:2

All other fields are reserved. B_0 ($SLBE_{37}$) is treated as a reserved field.

Figure 6.9: SLB Entry

Instructions cannot be executed from a No-execute (N=1) segment.

Segments may contain a mixture of page sizes. The L and LP bits specify the base virtual page size for the segment. The $SLB_{L||LP}$ encodings are those shown in Figure 6.10. The base virtual page size (also referred to as the “base page size”) is the smallest virtual page size that can be used to map a given access, and in most cases is the smallest virtual page size for the segment. (The exception is that multiple base virtual page sizes can occur within the same segment when the base page size specified for a given implicit access (e.g. of one segment table) does not match the base page size specified for another implicit access (e.g. of a different segment table or the process table) or for explicit accesses. References to the base page size for a segment will be understood not to preclude or functionally conflict with this possibility.) The base virtual page size is 2^b bytes. The actual virtual page size (also referred to as the “actual page size” or “virtual page size”) is specified by $PTE_{L\ LP}$.

encoding	base page size
0b000	4 KB
0b101	64 KB
additional values[1]	2^b bytes, where b $>$ 12; b may differ among encoding values

[1]The “additional values” are implementation-dependent, as are the corresponding base virtual page sizes. Any values that are not supported by a given implementation are reserved in that implementation.

Figure 6.10: 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.
  - $ESID_{24:35}$
  - $VSID_{38:49}$

  The bits in the above two items are ignored by the hardware.

The Class field of the SLBE is used in conjunction with the ***slbie***, ***slbieg***, and ***slbia*** instructions (see Section 6.9.3.2). “Class” refers to a grouping of SLB entries and implementation-specific lookaside information so that only entries in a certain group need be invalidated and others might be preserved. The Class value assigned to an implementation-specific lookaside entry derived from an SLB entry must match the Class value of that SLB entry. The Class value assigned to an implementation-specific lookaside entry derived from real mode address “translation,” or translations required to access the Segment Table Entry Group is 0.

Software must ensure that the SLB contains at most one entry that translates a given effective address, and that if the SLB contains an entry that translates a given effective address, then any previously existing translation of that effective address has been invalidated. An attempt to create an SLB entry that violates this requirement may cause a Machine Check.

Programming Note

Class values should be assigned such that Class 0 is used for translations that are expected to be long-lived and Class 1 is used for translations that are expected to be short-lived. This assignment facilitates use of the ***slbia*** instruction, for which several IH values cause preferential invalidation of Class 1 SLB entries and lookaside information entries.

Programming Note

It is permissible for software to replace the contents of a valid SLB entry without invalidating the translation specified by that entry provided the specified restrictions are followed. See Chapter 13 Note 10.

6.7.8.2 SLB Search

When the hardware searches the SLB, all entries are tested for a match with the EA. For a match to exist, the following conditions must be satisfied for indicated fields in the SLBE.

- V=1
- $ESID_{0:63-s}$=$EA_{0:63-s}$, where the value of s is specified by the B field in the SLBE being tested

If no match is found, the search fails. If one match is found, the search succeeds. If more than one match is found, one of the matching entries is used as if it were the only matching entry, or a Machine Check occurs.

If the SLB search succeeds, the virtual address (VA) is formed from the EA and the matching SLB entry fields as follows.

$$VA=VSID_{0:77-s} \,||\, EA_{64-s:63}$$

The Virtual Page Number (VPN) is bits 0:77-p of the virtual address. The value of p is the actual virtual page size specified by the PTE used to translate the virtual address (see Section 6.7.9.1). If $SLBE_N$ = 1, the N (No-execute) value used for the storage access is 1.

Programming Note

On an instruction fetch, if $SLBE_N$=1, a [Hypervisor] Instruction Storage interrupt may occur without the Page Table being searched.

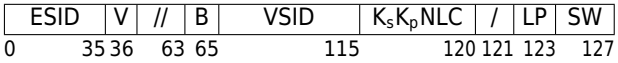
If the SLB search fails and the state is not such that a Segment Table search will be performed, a *segment fault* occurs. This is an Instruction Segment exception or a Data Segment exception, depending on whether the effective address is for an instruction fetch or for a data access.

Engineering Note

Treating an SLB entry with an unsupported value in the L||LP field as if $SLBE_V$ = 0 when translating effective addresses facilitates debugging of software. However, specific system compatibility requirements for migrating software may dictate a different treatment.

6.7.8.3 Segment Table Description and Search

The Segment Table is an aligned structure composed of 16B segment descriptors organized into 128 byte Segment Table Entry Groups (STEGs). Let q = STABSIZE+12, $\log_2$(size of the Segment Table). The base of the Segment Table in virtual address space is $STABORG_{0:77-q} \,||\, {}^{q}0$. Software must set the low order q-12 bits of 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.



Bit(s)	Name	Description
0:35	ESID	Effective Segment ID
36	V	Entry valid (V=1) or invalid (V=0)
64:65	B	Segment Size Selector 0b00 - 256 MB (s=28) 0b01 - 1 TB (s=40) 0b10 - reserved 0b11 - reserved
66:115	VSID	Virtual Segment ID
116	K_s	Supervisor (privileged) state storage key (see Section 6.7.13.2)
117	K_p	Problem state storage key (See Section 6.7.13.2.)
118	N	No-execute segment if N=1
119	L	Virtual page size selector bit 0
120	C	Class
122:123	LP	Virtual page size selector bits 1:2
124:127	SW	available for software use

All other fields are reserved.

Figure 6.11: Segment Table Entry

6.7.8.3.1 Primary Hash for 256MB Segment

The STEG is located at host VA $STABORG_{0:77-q} \,||\, EA_{43-q:35} \,||$ 0b0000000. Each of the 8 STEs are searched to find

a valid entry (V=1, B=0b00) that matches the ESID ($STE_{ESID[0:35]}$ = $EA_{0:35}$) of the access being translated.

6.7.8.3.2 Primary Hash for 1TB Segment

The STEG is located at host VA $STABORG_{0:77-q}$ || $EA_{31-q:23}$ || 0b0000000. Each of the 8 STEs are searched to find a valid entry (V=1, B=0b01) that matches the ESID ($STE_{ESID[0:23]}$ = $EA_{0:23}$) of the access being translated.

6.7.8.3.3 Secondary Hash for 256MB Segment

The STEG is located at host VA $STABORG_{0:77-q}$ || $\neg EA_{43-q:35}$ || 0b0000000. Each of the 8 STEs are searched to find a valid entry (V=1, B=0b00) that matches the ESID ($STE_{ESID[0:35]}$ = $EA_{0:35}$) of the access being translated.

6.7.8.3.4 Secondary Hash for 1TB Segment

The STEG is located at host VA $STABORG_{0:77-q}$ || $\neg EA_{31-q:23}$ || 0b0000000. Each of the 8 STEs are searched to find a valid entry (V=1, B=0b01) that matches the ESID ($STE_{ESID[0:23]}$ = $EA_{0:23}$) of the access being translated.

6.7.9 Hashed Page Table Translation

In Paravirtualized HPT mode, conversion of a 78-bit virtual address to a real address is done by searching the Page Table as shown in Figure 6.12.

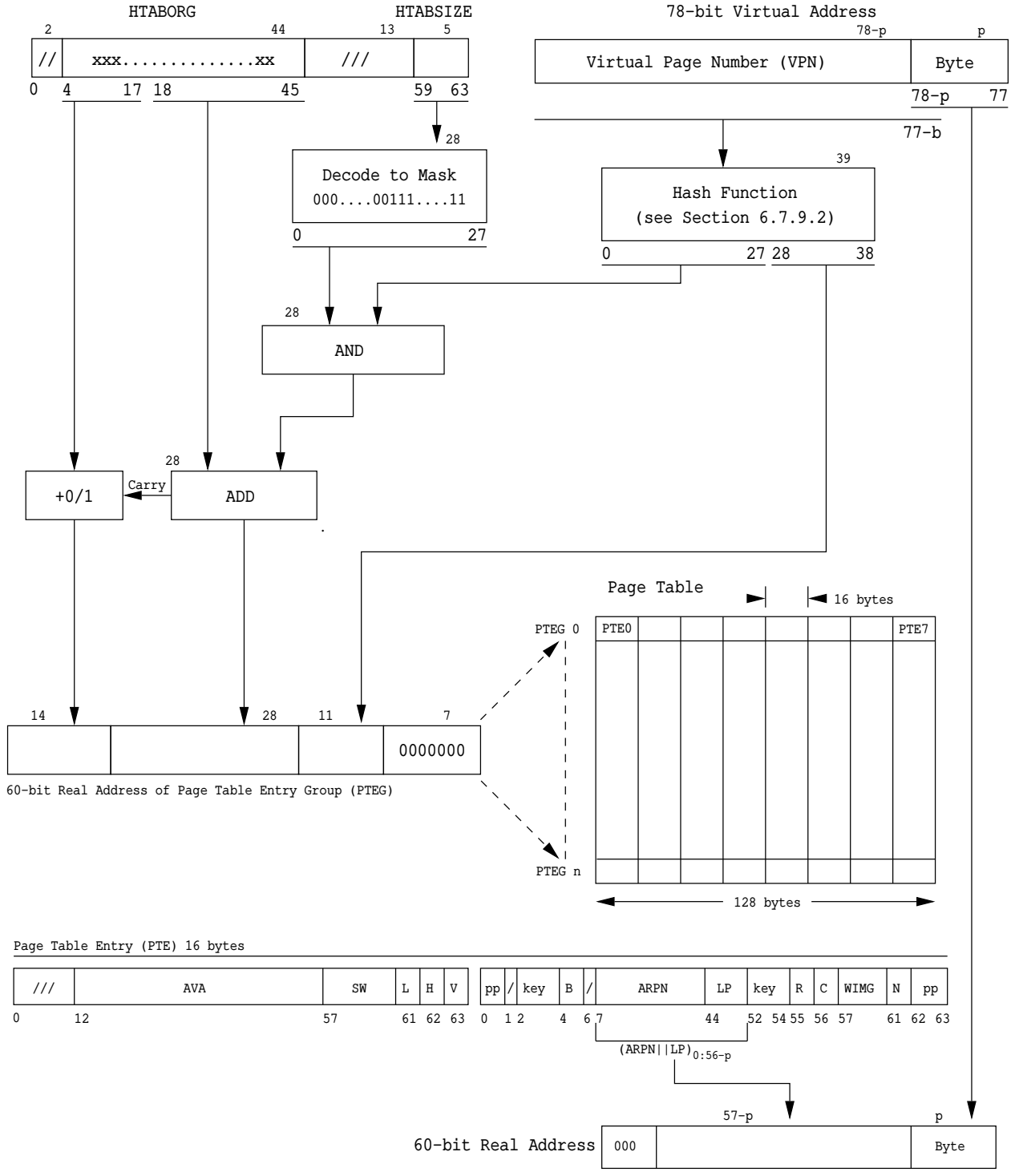


Figure 6.12: Translation of 78-bit virtual address to 60-bit real address

6.7.9.1 Hashed Page Table

The Hashed Page Table (HTAB) is a variable-sized data structure that specifies the mapping between virtual page numbers and real page numbers, where the real page number of a real page is bits 0:47 of the address of the first byte in the real page. The HTAB's size can be any size 2^n bytes where $18 \leq n \leq 46$. The HTAB must be located in storage having the storage control attributes that are used for implicit accesses to it (see Section 6.7.3.4). The starting address must be a multiple of 2^{18} bytes.

The HTAB contains Page Table Entry Groups (PTEGs). A PTEG contains 8 Page Table Entries (PTEs) of 16 bytes each; each PTEG is thus 128 bytes long. PTEGs are entry points for searches of the Page Table.

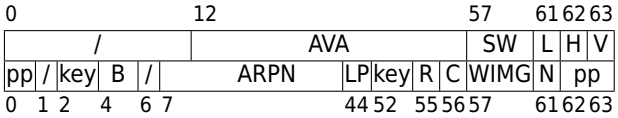
See Section 6.10 for the rules that software must follow when updating the Page Table.

Programming Note

The Page Table must be treated as a hypervisor resource (see Chapter 2), and therefore must be placed in real storage to which only the hypervisor has write access. Moreover, the contents of the Page Table must be such that non-hypervisor software cannot modify storage that contains hypervisor programs or data.

Page Table Entry

Each Page Table Entry (PTE) maps one VPN to one RPN. Figure 6.13 shows the layout of a PTE. This layout is independent of the Endian mode of the thread.



DW	Bit(s)	Name	Description
0	12:56	AVA	Abbreviated Virtual Address
	57:60	SW	Available for software use
	61	L	Virtual page size 0b0 - 4 KB 0b1 - greater than 4KB (large page)
	62	H	Hash function identifier
	63	V	Entry valid (V=1) or invalid (V=0)
1	0	pp	Page Protection bit 0
	2:3	key	KEY bits 0:1
	4:5	B	Segment Size 0b00 - 256 MB 0b01 - 1 TB 0b10 - reserved 0b11 - reserved
	7:43	ARPN	Abbreviated Real Page Number
	44:51	LP	Large page size selector
	52:54	key	KEY bits 2:4
	55	R	Reference bit
	56	C	Change bit
	57:60	WIMG	Storage control bits
	61	N	No-execute page if N=1
	62:63	pp	Page Protection bits 1:2

All other fields are reserved.

Figure 6.13: Page Table Entry

Programming Note

The H bit in the Page Table Entry should not be set to one unless the secondary Page Table search has been enabled.

Because the length of the Abbreviated Virtual Address (AVA) field is only 45 bits, on implementations of this version of the architecture the virtual address size cannot exceed 68 bits ($n \leq 68$). On implementations for which $n<68$, bits 0:67-n of the AVA field must be zeros.

If $b \leq 23$, the AVA field contains bits 10:54 of the VA. Otherwise bits 0:67-b of the AVA field contain bits 10:77-b of the VA, and bits 68-b:44 of the AVA field must be zero.

Programming Note

The AVA field omits the low-order 23 bits of the VA. These bits are not needed in the PTE, because the low-order b of these bits are part of the byte offset into the virtual page and, if $b<23$, the high-order 23-b of these bits are always used in selecting the PTEGs to be searched (see Section 6.7.9.2).

A virtual page is mapped to a sequence of 2^{p-12} contiguous real pages such that the low-order p-12 bits of the real page number of the first real page in the sequence are 0s.

$PTE_{L\ LP}$ 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 PTE_L=0, the base virtual page size and actual virtual page size are 4KB, and ARPN concatenated with LP (ARPN||LP) contains the page number of the real page that maps the virtual page described by the entry.

If PTE_L=1, the base page size and actual page size are specified by PTE_{LP}. In this case, the contents of PTE_{LP} have the format shown in Figure 6.14. 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 PTE_L=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.

PTE_{LP}	actual page size
r r r r _ r r r z	≥8 KB
r r r r _ r r z z	≥16 KB
r r r r _ r z z z	≥32 KB
r r r r _ z z z z	≥64 KB
r r r z _ z z z z	≥128 KB
r r z z _ z z z z	≥256 KB
r z z z _ z z z z	≥512 KB
z z z z _ z z z z	≥1 MB

Figure 6.14: Format of PTE_{LP} when PTE_L=1

There are at least 2 formats of PTE_{LP} that specify a 64 KB page. One format is used with $SLBE_{L||LP}$ = 0b000 and one format is used with $SLBE_{L||LP}$ = 0b101.

The actual page size selected by the LP field must not exceed the segment size selected by the B field. Forms of PTE_{LP} 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 PTE_{LP} format is at least $2^{12+(8-c)}$, where c is the number of r bits in the format.

Programming Note

Implementations often have TLBs and implementation-specific lookaside buffers (e.g. ERATs) used to cache translations of recently used storage addresses. Mapping virtual storage to large pages may increase the effectiveness of such lookaside buffers, improving performance, because it is possible for such buffers to translate a larger range of addresses, reducing the frequency that the Page Table must be searched to translate an address.

Engineering Note

For backward compatibility with implementations that comply with versions of the architecture that precede Version 2.03, implementations should support the 16 MB page size and should use $SLB_{L||LP}$=0b100 and PTE_{LP} = rrrr_rrr0 to specify that size. (In those versions there was no LP field in the SLBE or PTE. L=0 specified the 4 KB page size and L=1 specified the "large" page size, and the large page size supported by those implementations was 16 MB.)

If an implementation supports more than eight page sizes, it must also provide a means of mapping $SLB_{L||LP}$ to page sizes because the architecture permits at most eight page sizes to be in use concurrently. An implementation may also provide a means of mapping PTE_{LP} encodings to different pages sizes, which may be more feasible than encoding many pages sizes concurrently.

Treating a PTE with an unsupported value in the LP field as if PTE_V = 0 when translating effective addresses facilitates the debugging of software.

Engineering Note

Multiple schemes can be used for encoding the actual and base page sizes in PTE_{LP}. Only a subset of combinations of possible base and actual pages sizes can be encoded at one time. Ignoring backward compatibility with implementations that comply with earlier versions of the architecture, the following example shows how the LP field, when PTE_L=1, can be used to encode some combinations of base and actual page sizes. The LP field is split to convey that the leftmost bits are sometimes "r" (RPN) bits, some bits specify the actual page size given a particular base page size, and the rightmost bits specify the base page size and sometimes the actual page size.

PTE_{LP}			page size	
			actual	base
rrrrrr		00	16 KB	4 KB
rrrr	0	010	64 KB	4 KB
rr	011	010	256 KB	4 KB
	00111	010	16 MB	4 KB
	01111	010	16 GB	4 KB
rrrrrr		01	16 KB	16 KB
rrrr	0	011	64 KB	16 KB
rr	011	011	256 KB	16 KB
	00111	011	16 MB	16 KB
	01111	011	16 GB	16 KB
rrrr	0	110	64 KB	64 KB
rr	011	110	256 KB	64 KB
	00111	110	16 MB	64 KB
	01111	110	16 GB	64 KB
rr	0	00111	256 KB	256 KB
	001	00111	16 MB	256 KB
	011	00111	16 GB	256 KB
	001	01111	16 MB	16 MB
	011	01111	16 GB	16 MB
	011	11111	16 GB	16 GB

Instructions cannot be executed from a No-execute (N=1) page.

Page Table Size

The number of entries in the Page Table directly affects performance because it influences the hit ratio in the Page Table and thus the rate of page faults. If the table is too small, it is possible that not all the virtual pages that actually have real pages assigned can be mapped via the Page Table. This can happen if too many hash collisions occur and there are more than 16 entries for the same primary/secondary pair of PTEGs (when the secondary Page Table search is enabled) or more than 8 entries for the same primary PTEG (when the secondary Page Table search is disabled).

While this situation cannot be guaranteed not to occur for any size Page Table, making the Page Table larger than the minimum size (see Section 6.7.6.1) will reduce the frequency of occurrence of such collisions.

Programming Note

If large pages are not used, it is recommended that the number of PTEGs in the Page Table be at least half the number of real pages to be accessed. For example, if the amount of real storage to be accessed is 2^{31} bytes (2 GB), then we have $2^{31-12}=2^{19}$ real pages. The minimum recommended Page Table size would be 2^{18} PTEGs, or 2^{25} bytes (32 MB).

6.7.9.2 Page Table Search

When the hardware searches the Page Table, the accesses are performed as described in Section 6.7.3.4.

An outline of the HTAB search process is shown in Figure 6.12. Up to two hash functions are used to locate a PTE that may translate the given virtual address.

1. Primary Hash:

   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 $\log_2$(base page size specified in the SLBE that was used to translate the address).

   If s=28, the hash value is computed by Exclusive ORing $VA_{11:49}$ with $(^{11+b}0||VA_{50:77-b})$

   If s=40, the hash value is computed by Exclusive ORing the following three quantities: $(VA_{24:37}\ ||^{25}0)$, $(0||VA_{0:37})$, and $(^{b-1}0||VA_{38:77-b})$.

   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 ($LPCR_{TC}$=0), perform the secondary hash function as follows; otherwise do not perform step 2 and proceed to step 3 below.

   If s=28, the hash value is computed by taking the ones complement of the Exclusive OR of $VA_{11:49}$ with $(^{11+b}0||VA_{50:77-b})$

   If s=40, the hash value is computed by taking the ones complement of the Exclusive OR of the following three quantities: $(VA_{24:37}\ ||^{25}0)$, $(0||VA_{0:37})$, and $(^{b-1}0||VA_{38:77-b})$

   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 = minimum(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 (PTE_L = 0) or (PTE_{LP} specifies the 4KB base page size)
   - if b $\neq$ 12 then (PTE_L = 1) and (PTE_{LP} specifies the base page size specified by $SLBE_{L||LP}$)

   If no match is found, the search fails. The result is a page fault – a [Hypervisor] Instruction Storage exception or a [Hypervisor] Data Storage exception, depending on whether the effective address is for an instruction fetch or for a data access. If one match is found, the search succeeds. If more than one match is found, one of the matching entries is used as if it were the only matching entry, or a Machine Check occurs.

If the Page Table search succeeds, the real address (RA) is formed by concatenating the following values, where the p value is the $\log_2$(actual page size specified by $PTE_{L\ LP}$).

- 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 page size in between. The TLB entry may omit the base and actual page sizes. When the TLB is searched during address translation, the set of VA bits that is used in the match check is determined by the scope of the TLB entry rather than by the base page size and the base page size may be omitted from the match check. 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. []

Programming Note

In versions of the architecture that precede Version 3.1B, the TLB entry was required to contain the base and actual page sizes and the match criteria for the TLB entry were identical to those for the Page Table Entry. This enabled the base page size specified by L||LP in the SLB Entry to select among multiple potential matching TLB entries whose match criteria differed only by base page size. The optionality of base page size in the TLB entry on processors that comply with Version 3.1B of the architecture or with any subsequent version requires that software invalidate all TLB entries mapping a virtual segment after changing L||LP in the Segment Table and/or SLB Entry for the segment and before creating any PTEs, for the segment, that contain the new base page size. Base page size continues to be part of the PTE match criteria and a base page size match is required to cache a PTE in the TLB. As a result, software may replace PTEs containing the old base page size lazily, rather than invalidating them along with the TLB Entries when the base page size for the segment is changed.

Programming Note

If PTE_L = 0, the actual page size (and base page size) are 4 KB. Otherwise the actual page size and base page size are specified by PTE_{LP}.

Since hardware searches the Page Table using a value of b equal to $\log_2$ (base page size specified in the SLBE that was used to translate the address) regardless of the actual page size, the hardware Page Table search will identify different PTEs for VAs in different 2^b-byte blocks of the virtual page if the actual page size is larger than the base page size. Therefore, there may need to be a valid PTE corresponding to each 2^b-byte block of the virtual page that is referenced. For an actual page size that is larger than 2^{23} (8 MB), the PTE_{AVA} will differ among some or all of these PTEs. Depending on the Page Table size, some or all of these PTEs may be in the same PTEG. Any such PTEs that are in the same PTEG will differ in the value of PTE_H or PTE_{AVA} or both.

All PTEs for the same virtual page should have the same values in the Page Protection, KEY, ARPN, WIMG, and N fields. A set of values from any one of the PTEs that maps the virtual page may be used for an access in the virtual page since lookaside buffer information may be used to translate the virtual address.

To avoid creating multiple matching PTEs, software should not create PTEs for each of two different virtual pages that overlap in the virtual address space. If the virtual page sizes differ, two virtual pages overlap if the values of virtual address bits 0:77-p for both virtual pages are the same, where 2^p is the actual virtual page size of the larger page.

Programming Note

Because a segment may contain pages of different sizes, the Page Table search uses the segment's base page size (which is the same for all virtual pages in the segment).

- The value of b used when searching the Page Table to identify the PTEGs to be checked for a match is $\log_2$(segment's base page size).
- A PTE (in the selected PTEGs) satisfies the Page Table search only if the base page size specified in the PTE is equal to the segment's base page size.

The matching PTE supplies the actual page size, 2^p; this value of p is used in forming the real address.

A virtual page of 2^p bytes in a segment with a base page size of 2^b bytes may be mapped by as many as $2^{(p-b)}$ PTEs.

Programming Note

To obtain the best performance, Page Table Entries should be allocated beginning with the first empty entry in the primary PTEG, or with the first empty entry in the secondary PTEG if the primary PTEG is full and the secondary Page Table search is enabled ($LPCR_{TC}$=0).

In Paravirtualized HPT mode, the N (No-execute) value used for the storage access is the result of ORing the N bit from the matching PTE with the N bit from the SLB entry that was used to translate the effective address.

6.7.10 Radix Tree Translation

Radix Tree translation uses a nested set of tables to map storage with increasing granularity. Although there is no requirement for an individual table to have uniform content, Page Directories generally contain pointers to other Page Directories or Page Tables (Page Directory Entries, PDEs), while Page Tables are the leaf tables that contain PTEs. Each Page Directory Entry and Page Table Entry in the Radix Tree is 8 bytes long. A Radix Tree root descriptor (RTRD) specifies the size of the address being translated, the size of the root table, and its location. RTRDs appear in variants of the Partition and Process Table Entries. (See Figures 6.4 and 6.5.) The Root Page Directory Size (RPDS) is specified as $\log_2$ (number of entries in the table). That number of bits is taken from the most significant end of the portion of the address being translated, as an index to choose an element in the Root Page Directory. The entries in the Root Page Directory each point to another page of entries, and give its size in the Next Level Size field, PDE_{NLS}. 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. An example with RPDS = 13 and PDE_{NLS} = 9 in each Page Directory is shown in Figure 6.15.

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.

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 (see Section 6.7.10.3), the effective to guest real (process-scoped) translation may be cached, as well as the partition-scoped translation for each guest real address produced by the translation process. A minimum of two TLB accesses is required to complete a nested

translation: one for the effective to guest real address and one for the guest real to host real address. (An implementation may optimize the process, as long as the optimization can be managed correctly using the ***tlbie*** instructions that software will use to manage the logical model.) The scope of a TLB entry is the size of the page that is mapped by the corresponding PTE. When the TLB is searched during address translation, the scope of the TLB entry determines the set of address bits that is used in the match check; however, the page size need not be part of the match check.

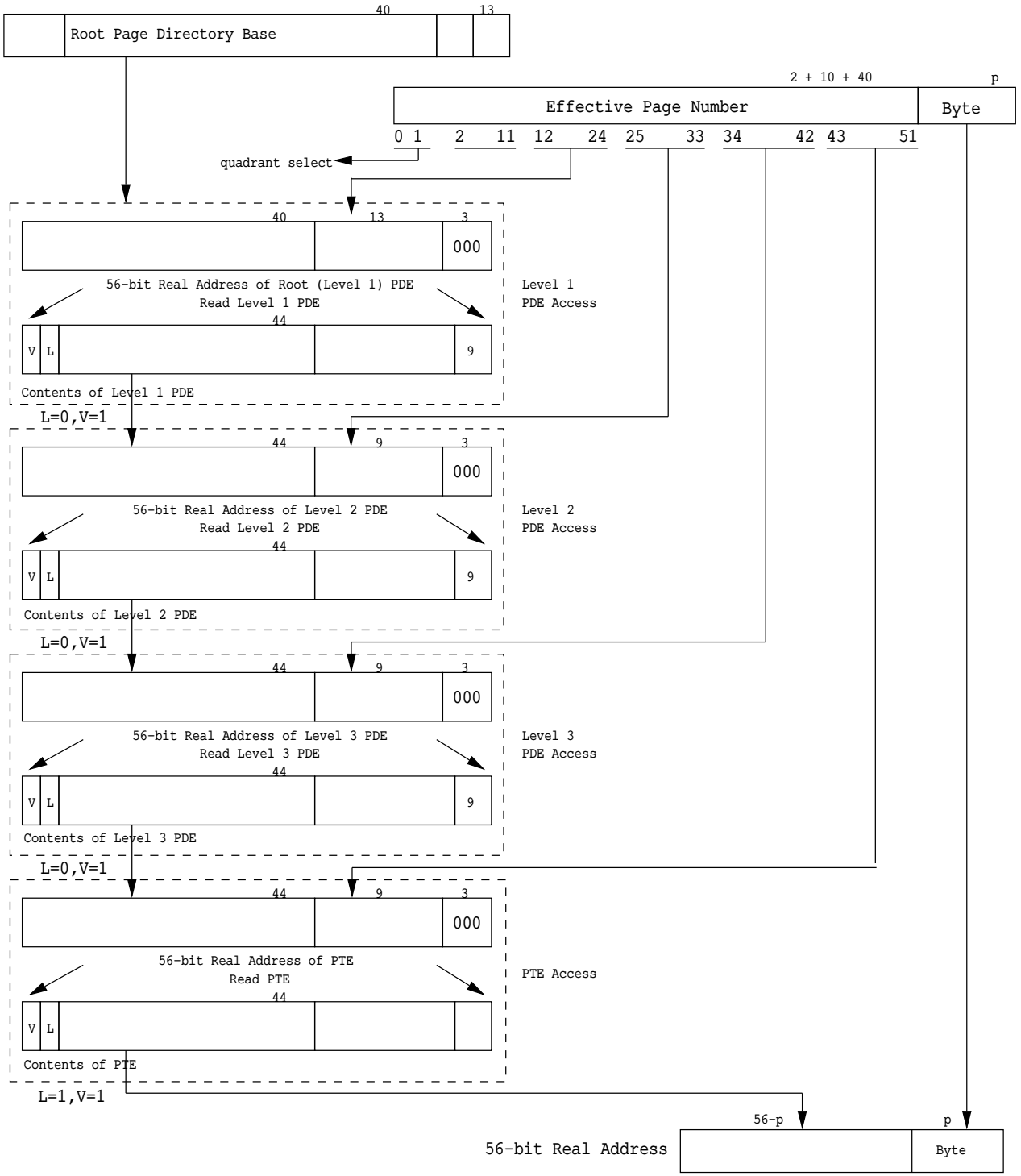
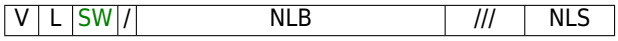


Figure 6.15: Four level Radix Tree walk translating a 52b EA with NLS=13 in the root PDE and NLS=9 in the other PDEs.

6.7.10.1 Radix Tree Page Directory Entry



Bit(s)	Name	Description
0	V	Valid
1	L	Leaf (entry is a PTE)
2	SW	Available for software use
4:55	NLB	Next Level Base
59:63	NLS	Next Level Size (size of next level of table is 2^{NLS+3}), NLS$\geq$5

All other fields are reserved.

Figure 6.16: Radix Tree Page Directory Entry

Programming Note

Although the SW field first appeared in the PDE in Version 3.1B of the architecture, it is available for software use on any implementation that supports Radix Tree translation.

Engineering Note

What address bits need to be implemented? There are a number of cases.

- For a PDE or PTE that produces a host real address, the size is determined by the size of the physical address space supported by the system.
- For a PTE or PDE that produces a guest real address with HR=1, the largest size is the Radix Tree Size indicated by RTS1||RTS2 for the partition-scoped translation or the system limit thereon.

6.7.10.2 Radix Tree Page Table Entry

Bit(s)	Name	Description
0	V	Valid
1	L	Leaf (entry is a PTE)
2	sw	SW bit 0
7:51	RPN	Real Page Number
52:54	sw	SW bits 1:3
55	R	Reference
56	C	Change
58:59	ATT	Attributes (equivalent WIMG value) 0b00 - normal memory (0010) 0b01 - reserved 0b10 - non-idempotent I/O (0111) 0b11 - tolerant I/O (0110)
60:63	EAA	Encoded Access Authority
	0	Privilege (applies only to process-scoped translation) 0 - problem state access permitted; privileged access controlled by key 0 of the [I]AMR 1 - privileged access only
	1	Read 0 - loads permitted only if EAA_2=1 1 - loads permitted
	2	Read/Write 0 - loads permitted only if EAA_1=1, stores not permitted 1 - loads and stores permitted
	3	Execute 0 - instruction execution not permitted 1 - instruction execution permitted

All other fields are reserved.

Figure 6.17: Radix Tree Page Table Entry

Architecture Note

In contrast to the more common, orthogonal "read" and "write" control bits in some architectures, EAA_1 and EAA_2 were defined to provide permissions that are the same as those provided by the pp bits in the HPT PTE. This has the advantage that the access authority encoding in an existing HPT-only TLB/ERAT design need not be changed to support Radix Tree translation, but the disadvantages of lack of write-only support and redundant meanings of $EAA_{1:2}$=0b11 and 0b01.

Programming Note

A future version of the architecture may change the meaning of $EAA_{1:2}$ = 0b01. In order to be compatible with such a change, software should encode read/write access permission using $EAA_{1:2}$ = 0b11 rather than $EAA_{1:2}$ = 0b01.

6.7.10.3 Nested Translation

When MSR_{HV}=0 and translation is enabled, each guest real address must undergo partition-scoped translation using the hypervisor's Radix Tree for the partition. See Figure 6.18.

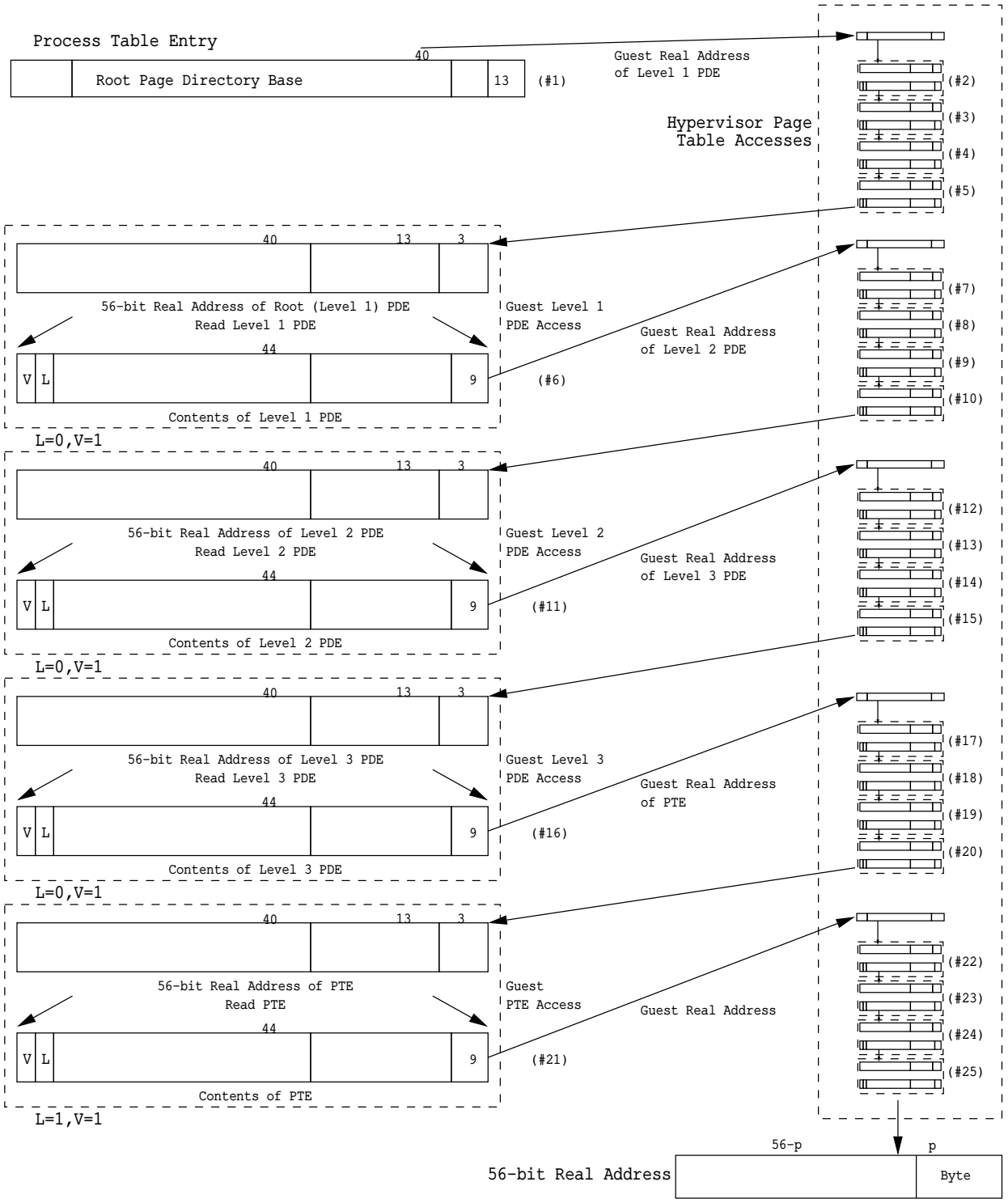


Figure 6.18: Radix on Radix Page Table search for a 52-bit EA depicting memory reads 1-24 numbered in sequence

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). [] The Guarded attribute is controlled by the process-scoped PTE. Mismatches of the Caching Inhibited attribute have the following behavior. If the process-scoped PTE specifies I=1 when the partition-scoped PTE specifies I=0, the result is I=0. The reverse mismatch causes a Data Storage or Instruction Storage exception, as appropriate for the access. The results of these rules are shown in Table 6.1. Together these rules can produce the WIMG=0b0011 state that any individual ATT value cannot express.

For instructions that cause a [Hypervisor] Data Storage interrupt if the storage operand is in Caching Inhibited storage, if I=1 in both the process-scoped and the partition-scoped PTE that map the storage operand the interrupt is a Data Storage interrupt. Similar applies to prefixed instructions that are in Caching Inhibited storage: if I=1 in both the process-scoped and the partition-scoped PTE that map the instruction, the interrupt caused by attempting to fetch the instruction is an Instruction Storage interrupt.

Programming Note

For the other cases in which access to storage with a particular storage control attribute causes a [Hypervisor] Data Storage or [Hypervisor] Instruction Storage interrupt the choice between the Hypervisor or the non-Hypervisor version of the interrupt is obvious.

- If the Radix Tree translation is not nested, the choice of interrupt is determined by the scope of the Radix Tree that is used.
- If the Radix Tree translation is nested and the attribute in question is the Guarded attribute, the interrupt is always the non-Hypervisor version because the G value used for the access is determined by the process-scoped PTE.

Unless otherwise stated or obvious from context, references elsewhere in the Books to storage control attributes for nested Radix Tree translations apply to the result of combining the guest and host storage control attributes as specified above. For example, the restrictions on the types of storage that can be accessed by AMOs in Section 4.5 of Book II apply to the results of the combining.

	partition-scoped ATT	00	10	11
process-scoped ATT	I/G	00	11	10
00	00	00	ATT conflict	ATT conflict
10	11	01	11	11
11	10	00	10	10

Table 6.1: Effective I and G attributes for nested translation

Programming Note

The mismatched Caching Inhibited attribute in which I=1 in the process-scoped PTE and I=0 in the partition-scoped PTE is given defined behavior instead of excepting in order to support frame buffer emulation. For frame buffer emulation, the guest believes it is writing to a frame buffer (I=1) in address space that the hypervisor maps to normal memory (I=0).

Reference and Change bit recording is done in both the process-scoped and partition-scoped Page Table Entries. Recording is done as described in Section 6.7.12, “Reference and Change Recording”.

6.7.11 Translation Process

As previously described, in its most complicated form the translation process includes the following steps:

- use of the PTCR to find the required Partition Table Entry
- use of the Partition Table Entry to find the partition-scoped Page Table
- use of the Partition Table Entry and the partition-scoped Page Table to find the required Process Table Entry
- use of the Process Table Entry and partition-scoped Page Table to find the required Segment Table Entry or walk the process-scoped Page Table (i.e. translate the effective address to a virtual or guest real address), and
- use of the partition-scoped Page Table to translate the virtual or guest real address.

Depending on the translation mode and process state, some of these steps may be skipped. The following subsections enumerate the cases and explain the steps in more detail.

6.7.11.1 Fully-Qualified Address

The storage control facilities enable hardware to perform the entire translation process given a “fully-qualified address” and context that makes it a unique input. In addition to its normal use, the term “effective address” is

sometimes used as shorthand for the fully-qualified address, and the architecture should be read with this possibilty in mind. The following are the components of the fully-qualified address.

- effLPID
- effPID
- EA

The additional context required to perform a translation or match a cached translation may include the following.

- $PATE_{HR}$ (selected using the value in LPIDR, not effLPID)
- $MSR_{HV\ PR\ IR\ DR}$

The translation mode is selected by the Host Radix bit found in the Partition Table Entry. The Host Radix bit indicates whether the partition is using HPT or Radix Tree translation. Given the overall process, $MSR_{HV\ PR\ IR\ DR}$ determine where and how the process is entered.

6.7.11.2 Finding the Page Tables

[The following description assumes that no legacy mode is active, i.e. $LPCR_{UPRT}$=1.]

The components of the fully-qualified address are used to determine the table(s) used in the translation process. The effective LPID and effective PID are used to find the appropriate Page Table base address(es) using the In-Memory Table structures. Some types of 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||^{12}0) in the PTCR with 16 times the effective LPID. The second doubleword of the entry contains the base address of the Process Table for the partition. The Process Table is assumed to be aligned in effective (HR=1, effLPID=0), virtual, or guest real address space. The Process Table Entry (PRTE) host real address is calculated by ORing the Process Table Base Address (PRTB||^{40}0 for for an HPT host and PRTB||^{12}0 for a radix host) in the PATE with 16 times the effective PID and then performing partition-scoped translation. (If the table is not aligned or is not large enough to support the PID value, an unreported error will most likely result.) The Process Table Entry at that location contains a process-scoped table base address, which is a guest real address for a radix guest (HV=0), a host real address for a radix host (HV=1), or a virtual address (all cases using HPT translation). The virtual or guest real address must be translated via the appropriate partition-scoped table.

Programming Note

The guest real or virtual address of the Process Table, for a radix or HPT guest, respectively, may be set via an hcall. The radix guest may choose to map the Process Table into its own effective address space. These matters are not visible to the architecture.

Programming Note

Note that the sole purpose of partition-scoped Page Table descriptor when LPID=0 for a radix host is to translate the effective addresses of the Process Table Entries for LPID=0. (If the Process Table Base address for LPID=0 was a real address, the Process Table would have to be in contiguous real storage.) This descriptor will commonly be the same as the descriptor found in the LPID=0, PID=0 Process Table Entry, both pointing to the hypervisor's own page trable, but it may be set up to point to a table used solely to translate the addresses of Process Table Entries.

Partition-scoped Page Table descriptors are found in the Partition Table as follows. The Partition Table Base Address is found in the PTCR. The effective LPID (times 16 bytes per partition) is used to index off the Partition Table Base Address to find the appropriate Partition Table Entry. The first doubleword of the entry contains the base address of the Page Table.

6.7.11.3 Obtaining Host Real Address, Radix on Radix

The following cases exist.

- Guest access to quadrant 0 with translation on: process-scoped translation is performed on LPIDR||PIDR||EA, with the result subject to partition-scoped translation with effective LPID=LPIDR.
- Guest OS 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$\neq$0.
- Hypervisor access to quadrant 2 with translation on: process-scoped translation is performed on LPIDR||0||EA, with the result subject to partition-scoped translation with effective LPID=LPIDR if LPIDR$\neq$0.
- Guest OS access with translation off: partition-scoped translation is performed with effective LPID = LPIDR.
- Hypervisor or host application access to quadrant 0 with LPIDR=0 and translation on: process-scoped translation is performed on 0||PIDR||EA.

- Hypervisor [] access to quadrant 3 with translation on: process-scoped translation is performed with 0||0||EA.
- Hypervisor or ultravisor real mode access: subject to EA_0 and either HRMOR or URMOR, as described in Section 6.7.3.1.

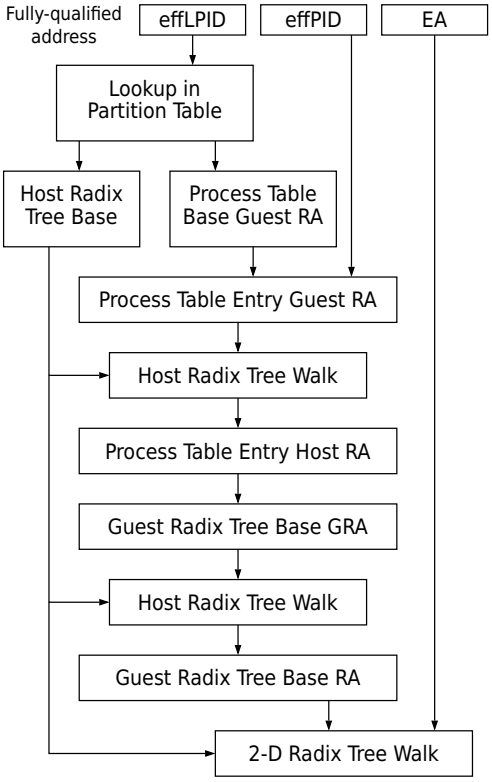


Figure 6.19: Radix on Radix translation, general case

6.7.11.4 Obtaining Host Real Address, HPT

There are two scenarios for Paravirtualized HPT translation. The first is the legacy scenario with a native HPT hypervisor. The second scenario is for a Radix Tree translation hypervisor providing a Paravirtualized HPT environment for the guest. In this latter scenario, the LPID=0 Partition Table Entry will have HR=1. For both scenarios the LPID value is always taken from LPIDR and the PID value is always taken from PIDR, even when MSR_{HV}=1. In the latter scenario, the hypervisor will explicitly set LPIDR=0 when it wants to use its Radix Tree(s).

When using Paravirtualized HPT translation, the process-scoped Page Tables are replaced by Segment Tables, and the description in Section 6.7.11.2, “Finding the Page Tables” can be read with that substitution in mind. The process-scoped translation is the effective-to-virtual translation described in Section 6.7.8. In-Memory Table walks are processed via the LPID=LPIDR partition-scoped HPT.

As with the previous enumerations, this is done from a hardware point of view. As a result, it does not differentiate the software cases for which Segment translation should only be satisfied by bolted translations.

The following cases exist.

- Guest access with translation on: process-scoped translation is performed on LPIDR||PIDR||EA with the result subject to partition-scoped translation using parameters from the matching segment descriptor.
- Hypervisor or adjunct access with translation on and LPID≠0: process-scoped translation, limited to an SLB search with no Segment Table walk, is performed on LPIDR||PIDR||EA, with the result subject to partition-scoped translation using parameters from the matching segment descriptor.
- Hypervisor or adjunct access with translation on and LPID=0: process-scoped translation (with Segment Table walk) is performed on LPIDR||PIDR||EA, with the result subject to partition-scoped translation using parameters from the matching segment descriptor.
- Guest OS access with translation off: subject to VPM, as described in Section 6.7.3.3.
- Hypervisor or ultravisor real mode access: subject to EA_0 and either HRMOR or URMOR, as described in Section 6.7.3.1.

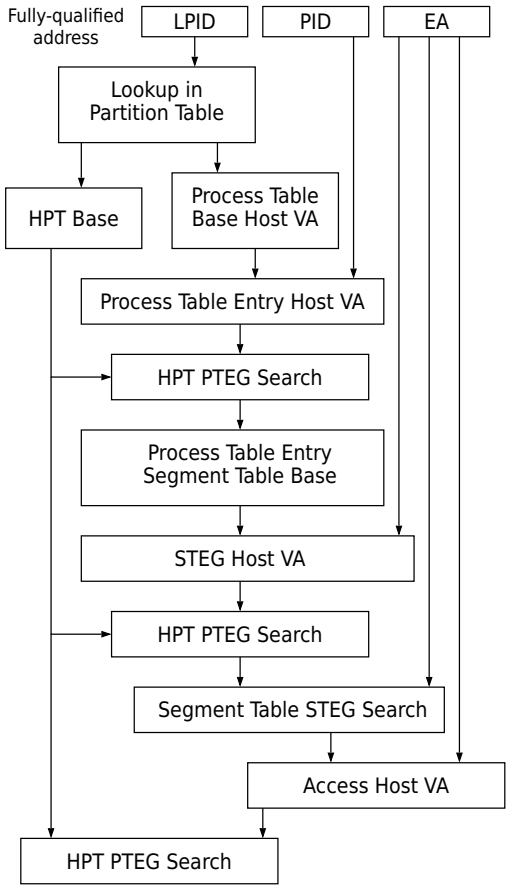


Figure 6.20: Paravirtualized HPT translation

6.7.12 Reference and Change Recording

When operating in Paravirtualized HPT mode, Reference (R) and Change (C) bits are updated in any one of what could be multiple (because of the multiple base size PTEs mapping a virtual page) Page Table Entries that map the virtual page that is being accessed. When operating in Radix on Radix mode, Reference (R) and Change (C) bits may be updated in multiple Page Table Entries that are accessed as part of the translation process. (For example, each access to a guest's Page Directory or Page Table Entry potentially sets a Reference bit in the partition-scoped table mapping it.) If the storage operand of a *Load* or *Store* instruction crosses a virtual page boundary, the accesses to the components of the operand in each page are treated as separate and independent accesses to each of the pages for the purpose of setting the Reference and Change bits.

For Radix Tree translation, hardware attempts to set the Reference and Change bits atomically, as though the PTE was read to perform the translation using a *Load And Reserve* instruction, and conditional on the translation being valid and correct (and on the existence of the reservation), the appropriate bit(s) are set as though with a *Store Conditional* instruction. ("as though" indicates that the reservation(s) held for this purpose are distinct from one another and from the reservation established by a *Load And Reserve* instruction.) If hardware is unable to set the bit(s) atomically, a [Hypervisor] Data Storage or [Hypervisor] Instruction Storage interrupt will be caused. For HPT translation, hardware sets the Reference and Change bits as though the PTE was read to perform the update using a (simple) *Load* instruction and the appropriate bit(s) are set as though with a (simple) *Store* instruction.

For both kinds of translation, setting the bits need not be atomic with respect to performing the access that causes the bits to be updated. For Radix Tree translation, the Reference bit must contain 1 in order to access the corresponding page and the Change bit must contain 1 in order to store to the corresponding page.

Programming Note

The interrupt indicates to software that it must set the appropriate bit(s) itself. Note that an instruction fetch can cause a Change bit to be set, for example in the host Page Table Entry that maps the guest Page Table Entry if the instruction fetch causes the Reference bit to be set in the guest Page Table Entry.

Programming Note

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. 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.

Engineering Note

When setting Reference and Change bits atomically, hardware must do the following:

1. read the PTE to be updated and establish a reservation covering the entire PTE,
2. validate that the PTE translates and permits the original (implicit or explicit) access that motivates the PTE update,
3. "OR" in the appropriate R/C bit(s), and
4. perform a conditional store based on the reservation established in step 1.
5. if the conditional store in step 4 fails, ideally repeat steps 1–4 a few times hoping for success, and eventually cause the appropriate [H]ISI or [H]DSI if the conditional store continues to fail.

Reference and Change bits are set by the hardware as described below. An attempt to access storage may cause one or more of the bits to be set (as described below) even if the access is not performed. The bits are updated in the Page Table Entry if the new value would otherwise be different from the old value for the virtual page, as determined by examining either the Page Table Entry or any lookaside information for the virtual page (e.g., TLB) maintained by the hardware.

Reference Bit

The Reference bit is set to 1 if the correspond-

ing access (load, store, implicit access, or instruction fetch) is required by the sequential execution model and is performed. Otherwise the Reference bit may be set to 1 if the corresponding access is attempted, either in-order or out-of-order, even if the attempt causes an exception, except that the Reference bit is not set to 1 for the access caused by an indexed *Move Assist* instruction for which the XER specifies a length of zero.

Change Bit

The Change bit is set to 1 if a *Store* instruction is executed and the store is performed or if an implicit update is performed. Otherwise in general the Change bit may be set to 1 if a *Store* instruction is executed and the store is permitted by the storage protection mechanism and, if the *Store* instruction is executed out-of-order, the instruction would be required by the sequential execution model in the absence of the following kinds of interrupts:

- system-caused interrupts (see Section 7.4 on page 1198)
- 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 a reserved function code, or if the instruction is a *Store Caching Inhibited* instruction executed when MSR_{DR}=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.

Programming Note

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 a reserved function code, *Store Caching Inhibited* executed when MSR_{DR}=1, 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.

Engineering Note

Any implementation-specific interrupt used to emulate instructions in software can be handled in a manner similar to a system-caused interrupt. That is, if the hardware can determine that the instruction to be emulated will not cause a precise architected interrupt then the Change bit can be set out-of-order past the instruction to be emulated under the same conditions as these bits can be set past a potential system-caused interrupt.

When the hardware updates the Reference and Change bits in a Page Table Entry, the accesses are performed as described in Section 6.7.3.4, “Storage Control Attributes for Implicit Storage Accesses” on page 1118. These Reference and Change bit updates are not necessarily immediately visible to software. Executing a ***sync*** instruction ensures that all Reference and Change bit updates associated with address translations that were performed, by the thread executing the ***sync*** instruction, before the ***sync*** instruction is executed will be performed with respect to that thread before the ***sync*** instruction’s memory barrier is created. There are additional requirements for synchronizing Reference and Change bit updates in multi-threaded systems; see Section 6.10, “Translation Table Update Synchronization Requirements” on page 1187.

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.

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 6.10 for the rules software must follow when updating Reference and Change bits.

Engineering Note

If the thread updates a Reference or Change bit in the Page Table Entry without using an atomic read/modify/write operation (i.e. when using HPT translation), care must be taken to avoid overwriting an update to any of these bits by another thread. Thus the datum written to the Page Table Entry must not contain a 0 value for any of these bits.

Subject to the preceding requirement, when the thread updates the Reference or Change bit in the Page Table Entry it is permissible to store the corresponding byte, halfword, word, or doubleword, with the relevant subset of these bits updated, from any lookaside information (e.g., TLB) maintained by the hardware.

Engineering Note

Since most TLB reloads do not require altering the Reference or Change bit in the Page Table Entry (PTE), it is suggested that on a TLB miss the search for the PTE be done without fetching the PTEs for exclusive access. This will reduce cache thrashing due to TLB reloads. It is assumed that a nonexclusive request for a PTE will be returned with exclusive access if no other thread has a copy.

Figure 6.21 on page 1145 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* and other instructions that are stated in the instruction descriptions to be treated as a *Load*, and similarly for "*Store-type*" instructions. 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".

Status of Access	**R**	**C**
Indexed *Move Assist* instruction with 0 length in XER	No	No
Load or *Store Atomic* instruction with reserved function code, *Load* or *Store Caching Inhibited* executed when MSR_{DR}=1	No	No
Storage protection violation	Acc[1]	No
Out-of-order Store-type inst'n, excluding ***dcbtst***		
Would be required by the execution model in the absence of system-caused or imprecise interrupts[3]	Acc	Acc[1 2]
All other cases	Acc	No
Out-of-order I-fetch or Load-type inst'n (including ***dcbtst***)	Acc	No
In-order *Load*-type or *Store*-type insn, access not performed[4]		
Store-type insn	Acc	Acc[2]
Load-type insn	Acc	No
Other in-order access		
Other ordinary *Store*, ***dcbz***, ***paste.***	Yes	Yes
icbi, ***icbt***, ***dcbt***, ***dcbtst***, ***dcbst***, ***dcbf***	Acc	No
I-fetch or ordinary *Load*, ***copy***, ***hashchk***[***p***]	Yes	No

"Acc" means that it is acceptable to set the bit.
1 It is preferable not to set the bit.
2 If C is set, R is also set unless it is already set.
3 For Floating-Point Enabled Exception type Program interrupts, "imprecise" refers to the exception mode controlled by $MSR_{FE0\ FE1}$.
4 This case does not apply to the *Touch* instructions, because they do not cause a storage access.

Figure 6.21: Setting the Reference and Change bits

6.7.13 Storage Protection

The storage protection mechanism provides a means for selectively granting instruction fetch access, granting read access, granting write access, and prohibiting access to areas of storage based on a number of control criteria.

The operation of the storage protection mechanism depends on the value of one or more of the following.

- MSR bits HV, S, IR, DR, PR
- the key bits and N bit in the associated SLB entry
- the page protection bits, key bits, N bit, and G attribute in the associated PTE
- the AMR, IAMR, AMOR, and UAMOR
- the Secure Memory property

The storage protection mechanism consists of the Virtual Page Class Key Protection mechanism described in Section 6.7.13.1, the Basic Storage Protection mechanism described in Section 6.7.13.2 and Section 6.7.13.3, the Radix Tree Translation Storage Protection mechanism described in Section 6.7.13.4, and the Secure Memory Protection mechanism described in Section 6.7.13.5.

In order for a storage access to be permitted, it must be permitted by all of the mechanisms that apply to it. If $SMFCTRL_E$=1, each storage access is subject to Secure Memory Protection independent of the translation mode of the access. In addition, each access is subject to other protection mechanisms depending on its translation mode, as listed below.

- MSR_{HV}=1 and address translation is disabled: Basic Storage Protection mechanism
- HR=0
  - access to instruction or data when address translation is enabled: Virtual Page Class Key Protection mechanism and Basic Storage Protection mechanism
  - all other cases (access to Process Table Entry or Segment Table Entry when address translation is enabled; access to instruction or data when MSR_{HV}=0 and address translation is disabled): Basic Storage Protection mechanism

Programming Note

Because the assumed K_s and K_p values are either 0 or irrelevant, these accesses are always permitted by the Basic Storage Protection mechanism.

- HR=1
  - access to instruction or data when address translation is enabled and effLPID≠0: Radix Tree Translation Storage Protection mechanisms of both the process-scoped and partition-scoped PTEs, except that the Guarded attribute (which affects storage protection for instruction fetches) is determined solely by the process-scoped PTE
  - access to instruction or data when address translation is enabled and effLPID=0: Radix Tree Translation Storage Protection mechanism of the process-scoped PTE
  - all other cases (access to Process Table Entry when address translation is enabled; access to process-scoped PDE or process-scoped PTE when address translation is enabled and effLPID≠0; access to instruction or data when MSR_{HV}=0 and address translation is disabled): Radix Tree Translation Storage Protection mechanism of the partition-scoped PTE

If an access associated with an instruction fetch is not permitted, an Instruction Storage exception or a Hypervisor Instruction Storage exception is generated. If an access associated with a data access is not permitted, a Data Storage exception or a Hypervisor Data Storage exception is generated. If HR=1, address translation is enabled, LPID≠0, and the access is not permitted by the process-scoped PTE, the exception is an Instruction Storage exception or Data Storage exception.

A *protection domain* is a maximal range of effective addresses, virtual addresses, or guest real addresses for which variables related to storage protection can be independently specified (including by default, as in virtual real, hypervisor real, and ultravisor real addressing modes), or a maximal range of addresses, effective, virtual, or guest real, for which variables related to storage protection cannot be specified. Examples include: a segment, a virtual page (including for the Virtualized Real Mode Area), the Virtualized Real Mode Area, the effective address range $0{:}2^{60}$-1 in hypervisor and ultravisor real addressing modes, and a maximal range of effective, virtual, or guest real addresses that cannot be mapped to real addresses. A *protection boundary* is a boundary between protection domains.

6.7.13.1 Virtual Page Class Key Protection

The Virtual Page Class Key Protection mechanism provides the means to assign virtual pages to one of 32 classes, and to modify data access permissions for each class by modifying the Authority Mask Register (AMR), shown in Figure 6.22, and to modify instruction access permissions for each class by modifying the Instruction Authority Mask Register (IAMR) shown in Figure 6.23.

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

Key0	Key1	Key2	. . .	Key29	Key30	Key31
0	2	4	6	58	60	62 63

Bits	Name	Description
0:1	Key0	Access mask for class number 0
2:3	Key1	Access mask for class number 1
. . .	. . .	. . .
2n:2n+1	Keyn	Access mask for class number n
. . .	. . .	. . .
62:63	Key31	Access mask for class number 31

Figure 6.22: 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.

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

Key0	Key1	Key2	. . .	Key29	Key30	Key31
0	2	4	6	58	60	62 63

Bits	Name	Description
0:1	Key0	Access mask for class number 0
2:3	Key1	Access mask for class number 1
. . .	. . .	. . .
2n:2n+1	Keyn	Access mask for class number n
. . .	. . .	. . .
62:63	Key31	Access mask for class number 31

Figure 6.23: 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 $IAMR_{2n+1}$ refers to the bit of the access mask corresponding to class number n.

- An instruction fetch is permitted if $IAMR_{2n+1}$=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 6.24 and Figure 6.25 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.

AMOR
0 63

Figure 6.24: Authority Mask Override Register (AMOR)

UAMOR
0 63

Figure 6.25: User Authority Mask Override Register (UAMOR)

The bits of the AMOR and UAMOR are in 1-1 correspondence with the bits of the AMR (i.e., [U]$AMOR_i$ corresponds to AMR_i). The AMOR affects modifications of the AMR and UAMOR in privileged but non-hypervisor state; the UAMOR affects modifications of the AMR in problem state.

Similarly, the odd bits of the AMOR are in 1-1 correspondence with the odd bits of the IAMR (i.e., $AMOR_{2j+1}$ corresponds to $IAMR_{2j+1}$). The AMOR affects modifications of the IAMR in privileged but non-hypervisor state; the IAMR cannot be accessed in problem state.

- When ***mtspr*** specifying the AMR (using either SPR 13 or SPR 29) or the IAMR is executed in privileged non-hypervisor state, the AMOR is used as a mask that controls which bits of the resulting AMR or IAMR contents come from register RS and which AMR or IAMR bits are not modified.
- Similarly, when ***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 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 5.4.4.

Software must ensure that both bits of each even/odd bit pair of the AMOR contain the same value—i.e., the contents of register RS for ***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.

Engineering Note

The implementation technique described above cannot be used for the AMR; all 64 bits of the AMR must be implemented.

Programming Note

When modifying the AMOR and/or UAMOR, the hypervisor should ensure that the two registers are consistent with one another before giving control to a non-hypervisor program. In particular, the hypervisor should ensure that if $AMOR_i$=0 then $UAMOR_i$=0, for all i in the range 0:63. (Having $AMOR_i$=0 and $UAMOR_i$=1 would permit problem state programs, but not the operating system, to modify AMR bit i.)

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 KEY_4 occupies the same bit in the PTE as the Data Address Compare mechanism's AC (Address Compare) bit, and that the definition of $ACCR_{62:63}$ is very similar to the definition of each even-odd pair of AMR bits. The technique is as follows, where PTE1 refers to doubleword 1 of the PTE.

- Set bits 2:3 and 62:63 of SPR 29 (which is either the ACCR or the AMR) to x, where x is the desired 2-bit value for controlling Data Address Compare matches, and set bits 0:1 to 0s.
- Set $PTE1_{54}$ (which is either the AC bit or KEY_4) to the same value that the AC bit would be set to, and set $PTE1_{2:3}$ (which are either RPN bits, that correspond to a real address size larger than the size supported by any implementation that supports the Data Address Compare mechanism, or $KEY_{0:1}$) and $PTE1_{52:53}$ (which are either reserved bits or $KEY_{2:3}$) to 0s.
- Use PTE_{KEY} values 0 and 1 only for purposes of emulating the Data Address Compare mechanism, except that PTE_{KEY} 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 PTE_{KEY} =31.
- When a Hypervisor Data Storage interrupt occurs, if $HDSISR_{42}$=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.)

Programming Note

An example of the use of the AMOR (and UAMOR) is to support adjuncts (see Section 6.7.4, "Definitions"), The hypervisor could use KEY value j for all data virtual pages that only the adjunct must be able to access. Before dispatching the partition for the first time, the hypervisor would initialize the three registers as follows.

AMR: all 0s except bits 2j and 2j+1, which would contain 1s
UAMOR: all 0s
AMOR: all 1s except bits 2j and 2j+1, which would contain 0s

Before dispatching the adjunct, the hypervisor would set UAMOR to all 0s, and would set the AMR to all 1s except bits 2j and 2j+1, which would be set to 0s. (Because the adjunct would run in problem state, there is no need for the hypervisor to modify the AMOR, and the adjunct cannot modify the UAMOR.) In addition, the hypervisor would prevent the partition from modifying or deleting PTEs that contain translations used by the adjunct.

(It may be desirable to avoid using KEY values 0, 1, and 31 for storage that only the adjunct can access, because these KEY values may be needed by the partition to emulate the Data Address Compare mechanism, as described above. Also, old software, that was written for an implementation that complies with a version of the architecture that precedes Version 2.04 (the version in which virtual page class keys were added), effectively uses KEY 0 for all virtual pages.)

Programming Note

Initialization of the UAMOR to all 0s, by the hypervisor before dispatching a partition for the first time, as described in the preceding Programming Note, permits operating systems (in partitions that run in a compatibility mode corresponding to Version 2.06 of the architecture or a subsequent version) to migrate gradually to supporting problem state access to the AMR — specifically, to avoid having to be changed immediately to modify the UAMOR and to save the AMR contents when an interrupt occurs from problem state. Relatedly, having the UAMOR contain all 0s while an application program is running protects old application programs that are "AMR-unaware". In the absence of programming errors, such application programs would not attempt to read or modify the AMR. However, having the UAMOR contain all 0s protects such programs against modifying the AMR inadvertently.

Permitting an "AMR-unaware" application program to modify the AMR (inadvertently) is potentially harmful for the obvious reasons. (The program might set to 1 an AMR bit corresponding to accesses that are necessary in order for the program to work correctly.) Moreover, even for an operating system that includes support for problem state modification of the AMR, having the UAMOR contain all 0s allows the operating system to avoid saving and restoring the AMR for "AMR-unaware" application programs. Such an operating system would provide a system service program that allows an application program to declare itself to be "AMR-aware" — i.e., potentially to need to modify the AMR. When an application program invokes this service, the operating system would set the UAMOR to the non-zero value appropriate to the access authorities (load and/or store, for one or more key values) that the application program is allowed to modify, and thereafter would save and restore the AMR (and preserve the UAMOR) for this application program. (Having the UAMOR contain all 0s does not prevent an "AMR-unaware" program from reading the AMR, but inadvertent reading of the AMR is likely to be much less harmful than inadvertently modifying it.)

[]

6.7.13.2 Basic Storage Protection, Address Translation Enabled

When address translation is enabled, , the Basic Storage Protection mechanism is controlled by the following.

- MSR_{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 6.7.8.1 and 6.7.9.2)
  - PTE_G, the G (Guarded) bit in the Page Table Entry used to translate the effective address

Using the above values, the following rules are applied.

1. For an instruction fetch, the access is not permitted if the N value is 1 or if PTE_G=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:

   $$Key \leftarrow (K_p \ \&\ MSR_{PR}) \mid (K_s \ \&\ \neg MSR_{PR})$$

   Using the computed Key, Figure 6.26 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”.

Key	PP	Access Authority
0	000	read/write
0	001	read/write
0	010	read/write
0	011	read only
0	110	read only
1	000	no access
1	001	read only
1	010	read/write
1	011	read only
1	110	no access
All PP encodings not shown above are reserved. The results of using reserved PP encodings are boundedly undefined.		

Figure 6.26: PP bit protection states, address translation enabled

6.7.13.3 Basic Storage Protection, Address Translation Disabled

When address translation is disabled, the Basic Storage Protection mechanism is controlled by MSR_{HV}, which (when MSR_{PR}=0) distinguishes between hypervisor state and privileged non-hypervisor state (see Chapter 2 and Section 6.7.3, “Ultravisor Real, Hypervisor Real, and Virtual Real Addressing Modes”). The following rules apply.

1. If MSR_{HV}=0, access authority is determined as described in Section 6.7.3.3.
2. If MSR_{HV}=1, the access is permitted.

6.7.13.4 Radix Tree Translation Storage Protection

For Radix Tree translation, an attempt to fetch instructions from Guarded storage is a storage protection violation. In all other respects, the storage protection mechanism for Radix Tree translation is completely different from what is provided for HPT translation. $EAA_{1:3}$ 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 6.17 and Figure 6.27 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 Privilege bit in the PTE is ignored for a privileged store. When bit 0 is 1, the Privilege bit must be 1 for a privileged store. Similarly when bit 1 is 0, the Privilege bit in the PTE is ignored for a privileged load. When bit 1 is 1, the Privilege 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 Privilege bit in the PTE is ignored for an attempt to execute the instruction in privileged state. When bit 1 is 1, the Privilege bit must be 1 to execute the instruction in privileged state.

Privilege (EAA_0)	Read (EAA_1)	Read/Write (EAA_2)	Execute (EAA_3)	Access Authority problem state (MSR_{PR}=1)	Access Authority privileged state (MSR_{PR}=0)
0	0	0	0	na	na
0	0	0	1	e	e*
0	0	1	0	rw	rw*
0	0	1	1	rwe	rwe*
0	1	0	0	r	r*
0	1	0	1	re	re*
0	1	1	0	rw	rw*
0	1	1	1	rwe	rwe*
1	0	0	0	na	na
1	0	0	1	na	e
1	0	1	0	na	rw
1	0	1	1	na	rwe
1	1	0	0	na	r
1	1	0	1	na	re
1	1	1	0	na	rw
1	1	1	1	na	rwe

```
Key:
na: no access
r : read
w : write
e : execute
* : For partition-scoped translation, including all translation of table entry addresses, all
    accesses in the entry are permitted.
    For process-scoped translation, each access in the entry is permitted if and only if the
    relevant bit of key 0 of the [I]AMR is 0.
```

Figure 6.27: Encoded Access Authority (aka page protection)

6.7.13.5 Secure Memory Protection

When $SMFCTRL_E$=1, Secure Memory Protection is enabled. Each location in main storage has a Secure Memory property mem_{SM}. mem_{SM}=1 indicates secure memory. mem_{SM}=0 indicates ordinary memory. Generally, only secure partitions and the ultravisor may access secure memory for explicit and implicit accesses. The one exception is that the Partition Table is commonly located in secure memory, but may be accessed implicitly as part of the translation process for software running with MSR_S=0. The granularity and method with which main storage is mapped for the Secure Memory property is implementation specific.

For each kind of access to a host real address that can cause a violation of Basic or Radix Tree Translation Storage Protection, a Secure Memory Protection exception is reported by the same type of interrupt as its Basic or Radix Tree Translation Storage Protection counterpart, except setting [H]DSISR or [H]SRR1 bit 43 instead of 36, as follows. For HPT translation, the exception is reported as an ISI or DSI if the thread is in hypervisor state, or if the thread is in non-hypervisor state when IR or DR is 1 for the appropriate type of access and VPM=0; otherwise as HISI or HDSI. For Radix Tree translation, the exception is reported as an ISI or DSI if effLPID=0; otherwise as HISI or HDSI. For HPT translation, the reporting approach described above is used for accesses which require translation but for which no Basic Storage Protection exception is possible. This includes accesses to the Segment Table Entry Group and Process Table Entry. []

In the preceding cases the host real address for the access is a result of address translation. A Secure Memory Protection exception can also be caused by accesses to a host real address that is not the result of address translation. (Such accesses cannot cause a violation of Basic or Radix Tree Translation Storage Protection.) These additional cases are reported as follows. For a hypervisor real mode access the exception is reported as an ISI or DSI. For a process-scoped radix tree access for effLPID=0 the exception is reported as an ISI or DSI. For a PTEG access the exception is reported as an ISI or DSI if $MSR_{HV\ PR}$=0b10; otherwise as HISI or HDSI. For a partition-scoped radix tree access the exception is reported as an HISI or HDSI unless effLPID=0, in which case the exception is reported as an ISI or DSI. These cases also set [H]DSISR or [H]SRR1 bit 43 to 1.

6.8 Storage Control Attributes

This section describes aspects of the storage control attributes that are relevant only to privileged software programmers. The rest of the description of storage control attributes may be found in Section 1.6 of Book II and subsections.

6.8.1 Guarded Storage

Storage is said to be "well-behaved" if the corresponding real storage exists and is not defective, and if the effects of a single access to it are indistinguishable from the effects of multiple identical accesses to it. Data and instructions can be fetched out-of-order from well-behaved storage without causing undesired side effects.

Storage is said to be Guarded if any of the following conditions is satisfied.

- MSR bit IR or DR is 1 for instruction fetches or data accesses respectively, or MSR_{HV}=0, and either G=1 or ATT=0b010 in the relevant Page Table Entry.
- MSR bit IR or DR is 0 for instruction fetches or data accesses respectively, MSR_{HV}=1, and the storage is specified by the Hypervisor Real Mode Storage Control facility to be treated as Guarded (see Section 6.7.3.2.1).

In general, storage that is not well-behaved should be Guarded. Because such storage may represent a control register on an I/O device or may include locations that do not exist, an out-of-order access to such storage may cause an I/O device to perform unintended operations or may result in a Machine Check.

The following rules apply to in-order execution of *Load* and *Store* instructions for which the first byte of the storage operand is in storage that is both Caching Inhibited and Guarded.

- *Load* or *Store* instruction that causes an atomic access

  If any portion of the storage operand has been accessed and an External, Decrementer, Hypervisor Decrementer, Performance Monitor, or Imprecise mode Floating-Point Enabled exception is pending, the instruction completes before the interrupt occurs.
- *Load* or *Store* instruction that causes an Alignment exception, or that causes a [Hypervisor] Data Storage exception for reasons other than Data Address Watchpoint match.

  The portion of the storage operand that is in Caching Inhibited and Guarded storage is not accessed.

  (The corresponding rules for instructions that cause a Data Address Watchpoint match are given in Section 10.4.)

6.8.1.1 Out-of-Order Accesses to Guarded Storage

In general, Guarded storage is not accessed out-of-order. The only exceptions to this rule are the following.

Load Instruction

If a copy of any byte of the storage operand is in a cache then that byte may be accessed in the cache or in main storage.

Instruction Fetch

If $MSR_{HV\ IR}$=0b10 then an instruction may be fetched if any of the following conditions are met.

1. The instruction is in a cache. In this case it may be fetched from the cache or from main storage.
2. The instruction is in a real page from which an instruction has previously been fetched, except that if that previous fetch was based on condition 1 then the previously fetched instruction must have been in the instruction cache.
3. The instruction is in the same real page as an instruction that is required by the sequential execution model, or is in the real page immediately following such a page.

Programming Note

Software should ensure that only well-behaved storage is copied into a cache, either by accessing as Caching Inhibited (and Guarded) all storage that may not be well-behaved, or by accessing such storage as not Caching Inhibited (but Guarded) and referring only to cache blocks that are well-behaved.

If a real page contains instructions that will be executed when MSR_{IR}=0 and MSR_{HV}=1, software should ensure that this real page and the next real page contain only well-behaved storage (or that the Hypervisor Real Mode Storage Control facility specifies that this real page is not Guarded).

6.8.2 Storage Control Bits

When the thread is not in hypervisor or ultravisor real addressing mode, each storage access is performed under the control of the Page Table Entry used to translate the effective address. Each Page Table Entry contains storage control bits that specify the presence or absence of the corresponding storage control for all accesses translated by the entry as shown in Figure 6.28 and Figure 6.29. In the following description, references to individual WIMG bits apply to the corresponding Radix ATT encoding, or to the result of combining the process-scoped and partition-scoped ATT encodings (see Section 6.7.10.3), except where otherwise stated or obvious from context.

Bit	Storage Control Attribute
W[1]	0 - not Write Through Required 1 - Write Through Required
I	0 - not Caching Inhibited 1 - Caching Inhibited
M[2]	0 - not Memory Coherence Required 1 - Memory Coherence Required
G	0 - not Guarded 1 - Guarded

[1] Support for the 1 value of the W bit is optional. Implementations that do not support the 1 value treat the bit as reserved and assume its value to be 0.

[2] Support for the 0 value of the M bit is optional, implementations that do not support the 0 value assume the value of the bit to be 1, and may either preserve the value of the bit or write it as 1.

[]

Figure 6.28: Storage control bits, HPT PTE

ATT value	Storage Type	
00	normal memory	(WIMG=0010)
01	reserved	
10	non-idempotent I/O	(WIMG=0111)
11	tolerant I/O	(WIMG=0110)
W=0 always for Radix Tree translation M=1 always for Radix Tree translation []		

Figure 6.29: Storage control bits, Radix PTE

When the thread is not in hypervisor or ultravisor real addressing mode, instructions are not fetched from storage for which the G bit in the Page Table Entry is set to 1; see Section 6.7.13.

When the thread is in hypervisor or ultravisor real addressing mode, the storage control attributes are implicit; see Section 6.7.3.2.

In Sections 6.8.2.1and 6.8.2.2, “access” includes accesses that are performed out-of-order, and references to W, I, M, and G bits include the values of those bits that are implied when the thread is in hypervisor or ultravisor real addressing mode.

Programming Note

In a system consisting of only a single-threaded processor which has caches, correct coherent execution does not require storage to be accessed as Memory Coherence Required, and accessing storage as not Memory Coherence Required may give better performance.

Engineering Note

Because instruction storage need not be consistent with data storage, it is permissible for an implementation to ignore the M bit for instruction fetches.

Treating instruction fetches as noncoherent may result in better performance in an implementation in which a coherent storage request has greater latency or overhead than a noncoherent storage request. However, care must be taken to avoid using a copy of a storage location that was fetched noncoherently (in response to an instruction fetch) to satisfy a subsequent coherent data request caused by a *Load*, *Store*, or *Cache Management* instruction. Also, care must be taken to ensure that the instruction sequence for instruction modification that is shown in Section 1.8 of Book II has the effects described there.

In system designs, consideration must be given to whether instruction fetches are to be noncoherent and, if so, how this choice affects the implementation of I/O subsystems and I/O caches. For example, if the implementation ignores the M bit for instruction fetches, the system could ensure that instructions being copied into main storage have been flushed from any I/O cache before the program using them is restarted.

6.8.2.1 Storage Control Bit Restrictions

Process- and partition-scoped ATT combinations that specify not Caching Inhibited for the process scope and Caching Inhibited for the partition scope are not permitted. See Table 6.1.

All combinations of W, I, M, and G values are permitted except those for which both W and I are 1. []

Programming Note

If an application program requests both the Write Through Required and the Caching Inhibited attributes for a given storage location, the operating system should set the I bit to 1 and the W bit to 0. []

At any given time, the value of the W bit must be the same for all accesses to a given real page.

At any given time, the value of the I bit must be the same for all accesses to a given real page.

6.8.2.2 Altering the Storage Control Bits

When changing the value of the W bit for a given real page from 0 to 1, software must ensure that no thread modifies any location in the page until after all copies of locations in the page that are considered to be modified in the data caches have been copied to main storage using ***dcbst*** or ***dcbf***[***l***] (or ***dcbstps*** or ***dcbfps***).

When changing the value of the I bit for a given real page from 0 to 1, software must set the I bit to 1 and then flush all copies of locations in the page from the caches using ***dcbf***[***l***] (or ***dcbfps***) and ***icbi*** before permitting any other accesses to the page. Note that similar cache management is required before using the Fixed-Point Load and Store Caching Inhibited instructions to access storage that has formerly been cached. (See Section 5.4.1 on page 1096.)

[]

Programming Note

It is recommended that ***dcbf*** be used, rather than ***dcbfl***, when changing the value of the I or W bit from 0 to 1. (***dcbfl*** would have to be executed on all threads for which the contents of the data cache may be inconsistent with the new value of the bit, whereas, if the M bit for the page is 1, ***dcbf*** need be executed on only one thread in the system.)

When changing the value of the M bit for a given real page, software must ensure that all data caches are consistent with main storage. The actions required to do this are system-dependent.

Programming Note

For example, when changing the M bit in some directory-based systems, software may be required to execute ***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 6.10.

6.9 Storage Control Instructions

6.9.1 Cache Management Instructions

This section describes aspects of cache management that are relevant only to privileged software programmers.

For a ***dcbz*** instruction that causes the target block to be newly established in the data cache without being fetched from main storage, the hardware need not verify that the associated real address is valid. The existence of a data cache block that is associated with an invalid real address (see Section 6.6) can cause a delayed Machine Check interrupt or a delayed Checkstop.

Each implementation provides an efficient means by which software can ensure that all blocks that are considered to be modified in the data cache have been copied to main storage before the thread enters any power conserving mode in which data cache contents are not maintained.

6.9.2 Synchronize Instruction

The *Synchronize* instruction is described in Section 4.6.3 of Book II, but only at the level required by an application programmer. This section describes properties of the instruction that are relevant only to operating system, hypervisor, and ultravisor software programmers.

The *Synchronize* instruction provides an ordering function for stores that are in set A of the memory barrier created by the *Synchronize* instruction, relative to data accesses caused by instructions that are executed on other threads after the occurrence of the interrupt that is caused by a ***msgsndp***, ***msgsnd***, or ***msgsndu*** instruction that follows the *Synchronize* instruction. The thread that is the target of the ***msgsndp***, ***msgsnd***, or ***msgsndu*** instruction is here called the "target thread".

- For ***msgsndp***, and L = 0, 1, or 2 (or 4 or 5) for the *Synchronize* instruction, the stores are performed with respect to the target thread before any data accesses caused by instructions that are executed on the target thread after the corresponding Directed Privileged Doorbell interrupt has occurred.
- For ***msgsnd*** or ***msgsndu***, and L = 0 or 2 (or 4) for the *Synchronize* instruction, [] the stores are performed with respect to any given other thread before any data accesses caused by instructions that are executed on the given thread after a ***msgsync*** instruction is executed on that thread after the corresponding Directed Hypervisor or Ultravisor Doorbell interrupt has occurred on the target thread.

Programming Note

Synchronize with L=1 (***lwsync***) or L=5 (***plwsync***) should not be used with ***msgsnd*** or ***msgsndu***. (If used, it will not have the desired ordering effect.)

Programming Note

The ***msgsync*** instruction, which is needed when ***msgsnd*** or ***msgsndu*** is used, is not needed when ***msgsndp*** is used because ***msgsndp*** targets only threads on the same multi-threaded processor as the thread executing the ***msgsndp***, while ***msgsnd*** and ***msgsndu*** can target any thread in the system. (If the target thread for ***msgsnd*** or ***msgsndu*** is on the same multi-threaded processor as the thread executing the ***msgsnd*** or ***msgsndu***, in principle the ***msgsync*** can be omitted. This optimization is practical only when the ***msgsnd***/***msgsndu*** topology is appropriately constrained, however, because the Directed Hypervisor or Ultravisor Doorbell interrupt provides no indication of which thread executed the ***msgsnd*** or ***msgsndu*** that caused the interrupt, so there is no easy way for the interrupt handler to determine whether the ***msgsync*** can be omitted.) ***msgsync*** is not needed or defined in V. 2.07 for a similar reason: ***msgsnd*** in V. 2.07 can target only threads on the same multi-threaded processor as the thread executing the ***msgsnd***.

The ordering done by ***sync*** (and ***phwsync*** and ***ptesync***) provides the appearance of "causality" across a sequence of ***msgsnd*** (or ***msgsndu***) instructions, as in the following example. "***msgsnd***->T1" means "***msgsnd*** instruction targetting 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=4 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.

Architecture Note

The requirement to order Reference and Change bit updates associated with *all* preceding address translations, rather than just the updates associated with instructions preceding the ***ptesync*** instruction – is needed only when software is altering the Page Table.

- 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.

Architecture Note

This property is necessary when changing the PTCR; see Note 3 in Chapter 13. It is not a corollary of the first property listed above, because memory barriers order storage accesses only with respect to other storage accesses.

- 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 ***slbieg***, ***slbiag***, 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 ***slbieg***, ***slbiag***, 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 ***slbie***, ***slbia***, and ***tlbiel*** 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*** insruction 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 1186.

  Similarly, in conjunction with the ***slbieg*** or ***slbiag*** 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 1171.

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 with SC≠0, or ***eieio***, or ***sync*** with L=4 or L=5 if appropriate).

Section 6.10, "Translation Table Update Synchronization Requirements" on page 1187 gives examples of uses of ***ptesync***.

6.9.3 Lookaside Buffer Management

All implementations have a Segment Lookaside Buffer (SLB). Independent of whether the executing partition operates in a mode that uses hardware SLB loading and bolting versus pure software loading (controlled by the value of $LPCR_{UPRT}$), software is responsible for keeping the SLB current with the segment mapping for the process that is executing. Proper management of the SLB across context switches is described in programming notes.

For performance reasons, most implementations also cache other information that is used in address translation. These caches may include: a Translation Lookaside Buffer (TLB) which is a cache of recently used Page Table Entries (PTEs); a cache of recently used translations of effective addresses to real addresses; a Page Walk Cache for Radix Tree translation; caching of the In-Memory Tables; or any combination of these. Lookaside information, including the SLB, is managed using the instructions described in the subsections of this section unless additional requirements are provided in implementation-specific documentation.

To simplify lookaside buffer management, hardware will only perform speculative translation for the context that is executing, in particular using the current effective values of LPID and PID. Except when LPIDR=0, no translations will be created and cached speculatively when HR=0 and MSR_{HV}=1. Furthermore, no translations will be created and cached speculatively in hypervisor or ultravisor real addressing mode. The limitation of speculative behavior in these situations is to cache a PATE when LPIDR is loaded and a PRTE when PIDR is loaded.

Programming Note

Speculative Segment Table walks are prohibited when MSR_{HV}=1 and LPIDR≠0 because adjunct translations are thread-specific and bolted.

Speculative Segment Table walks are allowed when MSR_{HV}=1 and LPIDR=0 to improve performance for "bare metal" operating systems (operating systems that run in hypervisor state). Bare metal operating systems would use LPIDR=0.

Engineering Note

Hardware will provide a means to disable speculative In-Memory Table walks as a safeguard against synchronization problems and other unforeseen difficulties.

Lookaside information derived from PTEs is not necessarily kept consistent with the Page Table. When software alters the contents of a PTE, in general it must also invalidate all corresponding TLB entries and implementation-specific lookaside information; exceptions to this rule are described in Section 6.10.1.2.

The effects of the ***slbie***, ***slbieg***, ***slbia***, ***slbiag***, and *TLB Management* instructions on address translations, as specified in Sections 6.9.3.2 for the SLB and 6.9.3.3 for the TLB, Page Walk Cache, and In-Memory Table caches, apply to all implementation-specific lookaside information that is used in address translation. Unless otherwise stated or obvious from context, references to SLB entry invalidation and TLB entry invalidation elsewhere in the Books apply also to invalidation of Page Walk Cache content, In-Memory Table cache content, and all implementation-specific lookaside information that is derived from SLB entries and PTEs, respectively.

All implementations provide a means by which software can invalidate all implementation-specific lookaside information that is derived from PTEs.

Implementation-specific lookaside information that contains translations of effective addresses to real addresses may include "translations" that apply in real addressing mode. Because such "translations" are affected by the contents of the LPCR, HRMOR, and URMOR, when software alters the (relevant) contents of these registers it must also invalidate the corresponding implementation-specific lookaside information. Software can invalidate all such lookaside information by using the ***slbia*** instruction with IH=0b000. 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. Also, unless otherwise stated or obvious from context, in material about TLB invalidations on other threads, "other threads" includes non-processor mechanisms that have synchronously-managed TLBs (e.g., nMMUs), and similarly for SLB invalidations.

Programming Note

Because the instructions used to manage TLBs, SLBs, Page Walk Caches, caches of Partition and Process Table Entries, and implementation-specific lookaside information may be changed in a future version of the architecture, it is recommended that software "encapsulate" their use into subroutines.

Programming Note

The function of all the instructions described in Sections 6.9.3.2 – 6.9.3.3 is independent of whether address translation is enabled or disabled.

For a discussion of software synchronization requirements when invalidating SLB and TLB entries, see Chapter 13.

Engineering Note

It is possible for the hardware to implement more than one TLB, such as one for data and one for instructions. If this approach is taken, the *TLB Management* instructions must affect all such TLBs as appropriate. Similar requirements apply to SLBs, Page Walk Caches, caches of Partition and Process Table Entries, and implementation-specific lookaside information that is used in address translation.

Engineering Note

The means by which software can invalidate implementation-specific lookaside information that is used in address translation should be independent of the contents of the information. Also, the time required for the invalidation should be reasonably small, and proportional to the amount of information being invalidated.

6.9.3.1 Thread-Specific Segment Translations

It is necessary to provide thread-specific temporary ESID to VSID translations. These translations cannot be placed

in valid entries in the Segment Table because the Segment Table has a process scope rather than a thread scope. Instead, software will use ***slbmte*** to install such translations in the SLB. All SLB entries created using ***slbmte*** are considered to be "software created." Software created entries will only translate accesses from the hardware thread by which they are installed. When $LPCR_{UPRT}$=1, they are also considered to be "bolted." Each thread has the ability to bolt four entries.

6.9.3.2 SLB Management Instructions

The only functionality described in this section that is relevant to Radix Tree translation is the use of ***slbia*** to invalidate implementation-specific lookaside information. The results of executing any other instruction in this section when HR=1 are boundedly undefined.

Software establishes translations in the SLB using ***slbmte***. 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 ***slbmte***. After updating a Segment Table Entry, software must use an ***slbie*** or ***slbieg*** instruction to remove lookaside information associated with the old contents of the entry. ***slbie*** may be used to invalidate software-created entries, but will not invalidate outboard translation caches. ***slbieg*** does not invalidate software-created entries, but, together with ***slbiag***, is the only way to invalidate outboard translation caches. When taking a PID or LPID out of service with the intent of reusing it, software should use ***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.) ***slbsync*** will establish order between ***slbieg*** and ***slbiag*** instructions and a subsequent ***ptesync***. ***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 ***slbia*** instruction to remove lookaside information associated with the old context. ***slbmfee*** and ***slbmfev*** may be used by the hypervisor to save software-created entries. ***slbmte*** is used to restore software-created entries. ***slbfee*** has no function when $LPCR_{UPRT}$=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 13.

SLB Invalidate Entry X-form

slbie RB

31	///	///	RB	434	/
0	6	11	16	21	31

```
ea0:35 ← (RB)0:35
if, for SLB entry that translates or
    most recently translated ea,
    entry_class = (RB)36 and
    entry_seg_size = size specified in (RB)37:38
then for SLB entry (if any) that translates ea
  SLBEV ← 0
  all other fields of SLBE ← undefined
else
   s ← log_base_2(entry_seg_size)
   esid ← (RB)0:63-s
   u ← undefined 1-bit value
   if u then
      if an SLB entry translates esid
         SLBEV ← 0
         all other fields of SLBE ← undefined
```

The operation performed by this instruction is based on the contents of register RB. The contents of this register are shown below.

RB

ESID	C	B	0s
0	36	37 39	63

$RB_{0:35}$	ESID
RB_{36}	Class
$RB_{37:38}$	B
$RB_{39:63}$	must be 0b0 || 0x000000

Let the Effective Address (EA) be any EA for which $EA_{0:35}$ = $(RB)_{0:35}$. Let the class be $(RB)_{36}$. Let the segment size be equal to the segment size specified in $(RB)_{37:38}$; the allowed values of $(RB)_{37:38}$, and the correspondence between the values and the segment size, are the same as for the B field in the SLBE (see Figure 6.9 on page 1127).

The class value and segment size must be the same as the class value and segment size in the SLB entry that translates the EA, or the values that were in the SLB entry that most recently translated the EA if the translation is no longer in the SLB; if these values are not the same, it is implementation-dependent whether the SLB entry (or implementation-dependent translation information) that translates the EA is invalidated, and the next paragraph need not apply.

If the SLB contains only a single entry that translates the EA, then that is the only SLB entry that is invalidated, except that it is implementation-dependent whether an implementation-specific lookaside entry for a real mode address “translation” is invalidated. If the SLB contains more than one such entry, then zero or more such entries are invalidated, and similarly for any implementation-specific lookaside information used in address translation; additionally, a machine check may occur.

SLB entries are invalidated by setting the V bit in the entry to 0, and the remaining fields of the entry are set to undefined values.

This instruction terminates any Segment Table walks being performed on behalf of the thread that executes it.

The hardware ignores the contents of RB listed below and software must set them to 0s.

- $(RB)_{37}$
- $(RB)_{39}$
- $(RB)_{40:63}$
- If s = 40, $(RB)_{24:35}$

If this instruction is executed in 32-bit mode, $(RB)_{0:31}$ must be zeros.

This instruction is privileged.

Special Registers Altered:

None

Programming Note

slbie does not affect SLBs on other threads.

Programming Note

The reason the class value specified by ***slbie*** must be the same as the Class value that is or was in the relevant SLB entry is that the hardware may use these values to optimize invalidation of implementation-specific lookaside information used in address translation. If the value specified by ***slbie*** differs from the value that is or was in the relevant SLB entry, these optimizations may produce incorrect results. (An example of implementation-specific address translation lookaside information is the set of recently used translations of effective addresses to real addresses that some implementations maintain in an Effective to Real Address Translation (ERAT) lookaside buffer.) Note that Radix Tree translations have no defined Class value, so frequent translation mode transitions may perform poorly under these optimizations.

When switching tasks in certain cases, it may be advantageous to preserve some implementation-specific lookaside entries while invalidating others. The ***slbia*** instruction specifying IH value 0b001 or 0b011 can be used for this purpose if SLB class values are appropriately assigned, i.e., a class value of 0 indicates that the entry should be preserved and a class value of 1 indicates the entry must be invalidated. Also, it is advantageous to assign a class value of 1 to entries that need to be invalidated via an ***slbie*** instruction while preserving implementation-specific lookaside entries that are derived from real mode address “translation” or translations required to access the Segment Table Entry Group, since such entries are assigned a class value of 0.

Programming Note

The B value in register RB may be needed for invalidating ERAT entries corresponding to the translation being invalidated.

Programming Note

When switching to execute an adjunct, a hypervisor will disable translation and use ***slbie*** to be sure there is no SLB entry mapping the effective address space that will be used by the incoming adjunct. It will then bolt an entry for the incoming adjunct and transfer control to that adjunct. While the thread is in hypervisor real addressing mode and during adjunct execution, no speculative Segment Table walks will be performed.

Architecture Note

The requirement that $RB_{0:31}$ contain zeros in 32-bit mode permits normal EA computation (in which the high-order 32 bits of the result are treated as zeros in 32-bit mode but not in 64-bit mode) to be used for ***slbie***.

Engineering Note

The requirement that the fields of the target SLBE, other than the V bit, are set to undefined values does not require any particular action by hardware. Hardware is permitted to write arbitrary values to the fields, to preserve the contents of the fields, etc. The fact that ***slbmfev*** and ***slbmfee*** return 0s for invalid SLBEs ensures that the contents of the invalid SLBEs cannot be used as a covert channel.

Engineering Note

On implementations that have an Effective to Real Address Translation lookaside buffer (ERAT), it is preferable that an ***slbie*** instruction not invalidate any ERAT entries for real mode address “translations”.

SLB Invalidate Entry Global X-form

slbieg RS,RB

31	RS	///	RB	466	/
0	6	11	16	21	31

```
target_PID = RS0:31
if MSRHV=1 then target_LPID = RS32:63
else target_LPID = LPIDR
ea0:35 ← (RB)0:35
for each thread with LPIDR=target_LPID and
                      PIDR=target_PID
  if, for each SLB entry that translates
      or most recently translated ea,
      entry_class = (RB)36 and
      entry_seg_size = size specified in (RB)37:38
  then for SLB entry (if any) that translates ea
           and is not software-created
    SLBEV ← 0
    all other fields of SLBE ← undefined
  else
     s ← log_base_2(entry_seg_size)
     esid ← (RB)0:63-s
     u ← undefined 1-bit value
     if u then
        if an SLB entry translates esid
            and the entry is not software-created
          SLBEV ← 0
          all other fields of SLBE ← undefined
```

The operation performed by this instruction is based on the contents of registers RS and RB. The contents of these registers are shown below.

RS

PID	LPID
0	32 63

RB

ESID	C	B	0s
0	36	37 39	63

$RS_{0:31}$	PID
$RS_{32:63}$	LPID
$RB_{0:35}$	ESID
RB_{36}	Class
$RB_{37:38}$	B
$RB_{39:63}$	must be 0b0 || 0x000000

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 class be $(RB)_{36}$. Let the segment size be equal to the segment size specified in $(RB)_{37:38}$; the allowed values of $(RB)_{37:38}$, and the correspondence between the values and the segment size, are the same as for the B field in the SLBE (see Figure 6.9 on page 1127).

Only SLBs for threads running on behalf of target_LPID and target_PID are searched. Software-created entries are not invalidated. The class value and segment size must be the same as the class value and segment size in the SLB entry that translates the EA, or the values that were in the SLB entry that most recently translated the EA if the translation is no longer in the SLB; if these values are not the same, it is implementation-dependent whether the SLB entry (or implementation-dependent translation information) that translates the EA is invalidated, and the next paragraph need not apply.

If the SLB contains only a single entry that translates the EA, then that is the only SLB entry that is invalidated, except that it is implementation-dependent whether an implementation-specific lookaside entry for a real mode address "translation" is invalidated. If the SLB contains more than one such entry, then zero or more such entries are invalidated, and similarly for any implementation-specific lookaside information used in address translation; additionally, a machine check may occur.

SLB entries are invalidated by setting the V bit in the entry to 0, and the remaining fields of the entry are set to undefined values.

The hardware ignores the contents of RB listed below and software must set them to 0s.

- $(RB)_{37}$
- $(RB)_{39}$
- $(RB)_{40:63}$
- If s = 40, $(RB)_{24:35}$

If this instruction is executed in 32-bit mode, $(RB)_{0:31}$ must be zeros.

The operation performed by this instruction is ordered by the ***eieio*** (or ***[p]hwsync*** or ***ptesync***) instruction with respect to a subsequent ***slbsync*** instruction executed by the thread executing the ***slbieg*** instruction. The operations caused by ***slbieg*** and ***slbsync*** are ordered by ***eieio*** as a fifth set of operations, which is independent of the other four sets that ***eieio*** orders.

This instruction is privileged except when $LPCR_{GTSE}$=0, making it hypervisor privileged.

Special Registers Altered:

None

Programming Note

slbieg does affect SLBs on other threads.

Programming Note

The reason the class value specified by ***slbieg*** must be the same as the Class value that is or was in the relevant SLB entry is that the hardware may use these values to optimize invalidation of implementation-specific lookaside information used in address translation. If the value specified by ***slbieg*** differs from the value that is or was in the relevant SLB entry, these optimizations may produce incorrect results. (An example of implementation-specific address translation lookaside information is the set of recently used translations of effective addresses to real addresses that some implementations maintain in an Effective to Real Address Translation (ERAT) lookaside buffer.) Note that Radix Tree translations have no defined Class value, so frequent translation mode transitions may perform poorly under these optimizations.

When switching tasks in certain cases, it may be advantageous to preserve some implementation-specific lookaside entries while invalidating others. The ***slbia*** instruction specifying IH value 0b001 or 0b011 can be used for this purpose if SLB class values are appropriately assigned, i.e., a class value of 0 indicates that the entry should be preserved and a class value of 1 indicates the entry must be invalidated. Also, it is advantageous to assign a class value of 1 to entries that need to be invalidated via an ***slbieg*** instruction while preserving implementation-specific lookaside entries that are derived from real mode address "translation" or translations required to access the Segment Table Entry Group, since such entries are assigned a class value of 0.

Programming Note

The B value in register RB may be needed for invalidating ERAT entries corresponding to the translation being invalidated.

Programming Note

Use of ***slbieg*** to invalidate software-created segment descriptors is a programming error. The architecture requires that bolted entries not be invalidated by the instruction.

Architecture Note

The requirement that $RB_{0:31}$ contain zeros in 32-bit mode permits normal EA computation (in which the high-order 32 bits of the result are treated as zeros in 32-bit mode but not in 64-bit mode) to be used for ***slbieg***.

Engineering Note

The requirement that the fields of the target SLBE, other than the V bit, are set to undefined values does not require any particular action by hardware. Hardware is permitted to write arbitrary values to the fields, to preserve the contents of the fields, etc. The fact that ***slbmfev*** and ***slbmfee*** return 0s for invalid SLBEs ensures that the contents of the invalid SLBEs cannot be used as a covert channel.

Engineering Note

On implementations that have an Effective to Real Address Translation lookaside buffer (ERAT), it is preferable that an ***slbieg*** instruction not invalidate any ERAT entries for real mode address "translations".

SLB Invalidate All X-form

slbia IH

31	//	IH	///	///	498	/
0	6	8	11	16	21	31

```
switch (IH)
  case (0b000, 0b001, 0b010, 0b110):
    for each SLB entry except SLB entry 0
      SLBE_V ← 0
      all other fields of SLBE ← undefined
  case (0b011):
    for each SLB entry such that SLBE_Class = 1
      SLBE_V ← 0
      all other fields of SLBE ← undefined
  case (0b100):
    for each SLB entry
      SLBE_V ← 0
      all other fields of SLBE ← undefined
  case (0b111):
```

slbia invalidates the contents of the SLB, and of implementation-specific lookaside information for effective to real address translations, based on the contents of the IH field as described below. SLB entries are invalidated by setting the V bit in the entry to 0. When an SLB entry is invalidated, the remaining fields of the entry are set to undefined values.

In the description of the IH values, "implementation-specific lookaside information" is shorthand for "implementation-specific lookaside information for effective to real address translations," and "when address translation was enabled" is shorthand for "when MSR_{IR} was equal to 1 or MSR_{DR} 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.

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 if it has a Class value of 0.
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 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.

All other IH values are reserved. If the IH field contains a reserved value, the set of SLB entries and implementation-specific lookaside information that is invalidated by the instruction is undefined.

When IH=0b000, 0b100, or 0b111, execution of this instruction has the side effect of clearing the storage access history associated with the Hypervisor Real Mode Storage Control facility. See Section 6.7.3.2.1, "Hypervisor Real Mode Storage Control" for more details.

This instruction terminates any Segment Table walks being performed on behalf of the thread that executes it, and ensures that any new table walks will be performed using the current PIDR value.

This instruction is privileged.

Special Registers Altered:

None

Extended Mnemonics:

Extended Mnemonics for *SLB Invalidate All*:

Extended mnemonic:	Equivalent to:
`slbia`	`slbia 0`

Programming Note

When performing a context switch between processes, an HPT operating system will use ***mtPIDR*** 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.)

Programming Note

Examples of the intended use of the IH values follow.

0b000 This setting should be used by a bare metal operating system or hypervisor to make extensive translation changes with address translation enabled and using $LPCR_{UPRT}$=0.

0b001 This setting should be used by an operating system that uses HPT translation and manages the Class bit but doesn't trust its use to manage the SLB (an extension to the longer-standing base function that could have compatibility implications) when switching tasks. Operating systems that use Radix Tree translation may also use this setting.

0b010 This setting should be used by an operating system that uses HPT translation and does not manage the Class bit when switching tasks.

0b011 This setting should be used by an operating system that uses HPT translation, manages the Class bit and trusts its use to manage the SLB when switching tasks. Operating systems that use Radix Tree translation may also use this setting.

0b100 This setting should be used by a bare metal operating system or hypervisor to make extensive translation changes with address translation disabled or using $LPCR_{UPRT}$=1.

0b110 This setting should be used by the hypervisor when switching partitions when $LPCR_{UPRT}$=0 and address translation is enabled.

0b111 This setting is provided mainly for use prior to product shipment, but may provide benefit in an environment that uses Radix Tree translation if SLB invalidation is much slower than ERAT invalidation.

Programming Note

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

Engineering Note

It is desirable to preserve SLB entries and ERAT entries that are not required to be invalidated. (For some IH values, certain SLB entries are required to be preserved.)

Engineering Note

The requirement that the fields of the target SLBE, other than the V bit, are set to undefined values does not require any particular action by hardware. Hardware is permitted to write arbitrary values to the fields, to preserve the contents of the fields, etc. The fact that ***slbmfev*** and ***slbmfee*** return 0s for invalid SLBEs ensures that the contents of the invalid SLBEs cannot be used as a covert channel.

SLB Invalidate All Global X-form

slbiag RS,L

31	RS	///	L	///	850	/
0	6	11	15	16	21	31

```
if L=0 then target_PID = RS0:31
if MSRHV=1 then target_LPID = RS32:63
else target_LPID = LPIDR
for each nest SLB
  for each SLBE with LPID=target_LPID
                 and (PID=target_PID | L=1)
    SLBEV ← 0
    all other fields of SLBE ← undefined
```

The operation performed by this instruction is based on the contents of register RS. The contents of this register is shown below.

RS

PID	LPID
0	32 63

$RS_{0:31}$ PID
$RS_{32:63}$ LPID

If L=0, let the target PID be $RS_{0:31}$. If the instruction is executed in hypervisor state, let the target LPID be $RS_{32:63}$; otherwise let the target LPID be the contents of LPIDR.

All nest SLBs are searched. If L=0, each SLBE for process PID in partition LPID is invalidated. If L=1, each SLBE for partition LPID is invalidated.

SLB entries are invalidated by setting the V bit in the entry to 0, and the remaining fields of the entry are set to undefined values.

All implementation specific lookaside information associated with SLB-derived translations for the target LPID || PID (L=0) or for the target LPID (L=1) is invalidated. Additional implementation specific lookaside information may be invalidated.

The operation performed by this instruction is ordered by the ***eieio*** (or [***p***]***hwsync*** or ***ptesync***) instruction with respect to a subsequent ***slbsync*** instruction executed by the thread executing the ***slbiag*** instruction. The operations caused by ***slbiag*** and ***slbsync*** are ordered by ***eieio*** as a fifth set of operations, which is independent of the other four sets that ***eieio*** orders.

This instruction is privileged except when $LPCR_{GTSE}$=0, making it hypervisor privileged.

Special Registers Altered:

None

Extended Mnemonics:

Extended Mnemonics for *SLB Invalidate All Global*:

Extended mnemonic:	Equivalent to:
`slbiag RS`	`slbiag RS,0`

Programming Note

slbiag does not affect SLBs on processor threads.

"g" (Global) in the name of the instruction reflects the fact that a future version of the architecture may extend the definition of ***slbiag*** to allow programs to specify additional sets of SLBs that the instruction affects, possibly including SLBs on processor threads.

Programming Note

slbiag serves as both a basic and an extended mnemonic. The Assembler will recognize an ***slbiag*** mnemonic with two operands as the basic form, and an ***slbiag*** mnemonic with one operand as the extended form. In the extended form the L operand is omitted and assumed to be 0.

Engineering Note

The requirement that the fields of the target SLBE, other than the V bit, are set to undefined values does not require any particular action by hardware. Hardware is permitted to write arbitrary values to the fields, to preserve the contents of the fields, etc. The fact that ***slbmfev*** and ***slbmfee*** return 0s for invalid SLBEs ensures that the contents of the invalid SLBEs cannot be used as a covert channel.

Engineering Note

On nest translation implementations that have an Effective to Real Address Translation lookaside buffer (ERAT), it is preferable that an ***slbiag*** instruction not invalidate any ERAT entries for real mode address "translations".

SLB Move To Entry X-form

slbmte RS,RB

31	RS	///	RB	402	/
0	6	11	16	21	31

When $LPCR_{UPRT}$=0, this instruction is the sole means for specifying Segment translations to the hardware. When $LPCR_{UPRT}$=1, Segment Table walks populate the SLB, and this instruction is used only to bolt thread-specific Segment translations.

The SLB entry specified by bits 52:63 of register RB is loaded from register RS and from the remainder of register RB. The contents of these registers are interpreted as shown below.

RS

B	VSID	K_sK_pNLC	0	LP	0s
0 2		52	57	58 60	63

RB

ESID	V	0s	index
0	36 37	52	63

$RS_{0:1}$	B
$RS_{2:51}$	VSID
RS_{52}	K_s
RS_{53}	K_p
RS_{54}	N
RS_{55}	L
RS_{56}	C
RS_{57}	must be 0b0
$RS_{58:59}$	LP
$RS_{60:63}$	must be 0b0000
$RB_{0:35}$	ESID
RB_{36}	V
$RB_{37:51}$	must be 0b000 || 0x000
$RB_{52:63}$	index, which selects the SLB entry

On implementations that support a virtual address size of only n bits, n<78, $(RS)_{2:79-n}$ must be zeros.

When $LPCR_{UPRT}$=1, the value of index must not exceed 3. $(RB)_{52:61}$ are ignored.

High-order bits of $(RB)_{52:63}$ that correspond to SLB entries beyond the size of the SLB provided by the implementation must be zeros.

The hardware ignores the contents of RS and RB listed below and software must set them to 0s.

- $(RS)_{57}$
- $(RS)_{60:63}$
- $(RB)_{37:51}$

If this instruction is executed in 32-bit mode, $(RB)_{0:31}$ must be zeros (i.e., the ESID must be in the range 0:15).

This instruction must not be used to load a segment descriptor that is in the Segment Table when $LPCR_{UPRT}$=1, and cannot be used to invalidate the translation contained in an SLB entry.

This instruction is privileged.

Special Registers Altered:

None

Programming Note

The reason ***slbmte*** must not be used to load segment descriptors that are in the Segment Table is that there could be a race condition with hardware loading the same segment descriptor, resulting in duplicate SLB entries. Software must not allow duplicate SLB entries to be created; see Section 6.7.8.2, “SLB Search”.

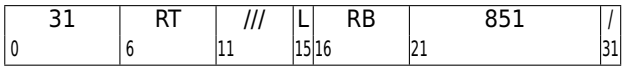
The reason ***slbmte*** cannot be used to invalidate an SLB entry is that it does not necessarily affect implementation-specific address translation lookaside information. ***slbie*** (or ***slbia***) must be used for this purpose.

Architecture Note

The requirement that $(RB)_{0:31}$ contain zeros in 32-bit mode permits normal EA computation (in which the high-order 32 bits of the result are treated as zeros in 32-bit mode but not in 64-bit mode) to be used for ***slbmte***.

SLB Move From Entry VSID X-form

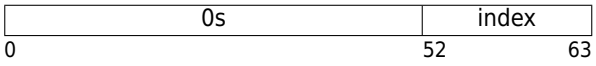
slbmfev RT,RB



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 below.

RT

RB



$RT_{0:1}$	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

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.

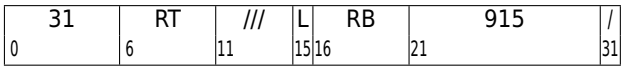
The use of the L field is implementation specific.

Special Registers Altered:

None

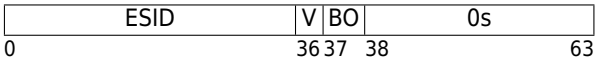
SLB Move From Entry ESID X-form

slbmfee RT,RB

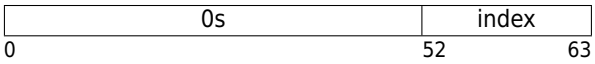


This instruction is used to read software-loaded SLB entries. When $LPCR_{UPRT}$=0, the entry is specified by bits 52:63 of register RB. When $LPCR_{UPRT}$=1, only the first four entries can be read, so bits 52:61 of register RB are ignored. If the specified entry is valid (V=1), the contents of the 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 below.

RT



RB



$RT_{0:35}$	ESID
RT_{36}	V
RT_{37}	BO, entry is bolted
$RT_{38:63}$	set to 0b000 || 0x00_0000
$RB_{0:51}$	must be 0x0_0000_0000_0000
$RB_{52:63}$	index, which selects the SLB entry

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

Engineering Note

On some implementations, the L bit may be used for debugging purposes, to enable entries other than the four bolted ones to be read. It is not specified in the architecture because the long term goal is to enable the non-bolted entries to be kept in a larger separate array that is not directly readable by software. The assembler should be slbmfev RT, RB, L and slbmfee RT, RB, L.

SLB Find Entry ESID X-form

slbfee. RT,RB

31	RT	///	RB	979	1
0	6	11	16	21	31

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 below.

RT

B	VSID	K_sK_pNLC	0	LP	0s
0 2		52	57	58 60	63

RB

ESID	0000	0s
0	36 40	63

$RT_{0:1}$	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
$RB_{0:35}$	ESID
$RB_{36:39}$	must be 0b0000
$RB_{40:63}$	must be 0x000000

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 $LPCR_{UPRT} \neq 0$, j=0b0.

$$CR0_{LT\ GT\ EQ\ SO} = 0b00 \mid\mid j \mid\mid XER_{SO}$$

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:

Register	*Field(s)*
CR	CR0

Architecture Note

The requirement that $RT_{60:63}$ be set to zero permits the SLB entry to be enlarged in the future if that proves desirable.

The requirement that $RB_{0:31}$ contain zeros in 32-bit mode permits normal EA computation (in which the high-order 32 bits of the result are treated as zeros in 32-bit mode but not in 64-bit mode) to be used for **slbfee.**.

Programming Note

When $LPCR_{UPRT}$=0, the hypervisor can use ***slbfee*** to save the contents of any SLBE that the partition has created to map an ESID that is needed by an adjunct, and later use the saved contents to restore the partition-created SLBE after the adjunct has completed execution. The hypervisor must also use ***slbie***, twice, first to invalidate the partition-created mapping and later to invalidate the adjunct's mapping.

When $LPCR_{UPRT}$=1, the partition's SLBE will be restored from the Segment Table by hardware, on demand, after the second ***slbie*** has been executed. There is no need for the hypervisor to save and restore the partition's SLBE and hence no need to use ***slbfee***.

When the need for $LPCR_{UPRT}$=0 has ended, ***slbfee*** may be removed from the architecture. Programs that run with $LPCR_{UPRT}$=1 should not use ***slbfee***.

SLB Synchronize X-form

slbsync

31	///	///	///	338	/
0	6	11	16	21	31

The ***slbsync*** instruction provides an ordering function for the effects of all ***slbieg*** and ***slbiag*** instructions executed by the thread executing the ***slbsync*** instruction, with respect to the memory barrier created by a subsequent ***ptesync*** instruction executed by the same thread. Executing a ***slbsync*** instruction ensures that all of the following will occur.

- All SLB invalidations caused by ***slbieg*** and ***slbiag*** instructions preceding the ***slbsync*** instruction will have completed on any other thread before any data accesses caused by instructions following the ***ptesync*** instruction are performed with respect to that thread.
- All storage accesses by other threads for which the address was translated using the translations being invalidated will have been performed with respect to the thread executing the ***ptesync*** instruction, to the extent required by the associated Memory Coherence Required attributes, before the ***ptesync*** instruction's memory barrier is created.

The operation performed by this instruction is ordered by the ***eieio*** (or [***p***]***hwsync*** or ***ptesync***) instruction with respect to preceding ***slbieg*** and ***slbiag*** instructions executed by the thread executing the ***slbsync*** instruction. The operations caused by ***slbieg*** or ***slbiag*** and ***slbsync*** are ordered by ***eieio*** as a fifth set of operations, which is independent of the other four sets that ***eieio*** orders.

The ***slbsync*** instruction may complete before operations caused by ***slbieg*** or ***slbiag*** instructions preceding the ***slbsync*** instruction have been performed.

This instruction is privileged except when $LPCR_{GTSE}$=0, making it hypervisor privileged.

See Section 6.10 for a description of other requirements associated with the use of this instruction.

Special Registers Altered:

None

Programming Note

slbsync should not be used to synchronize the completion of ***slbie***.

6.9.3.3 TLB Management Instructions

In addition to managing the TLB, ***tlbie*** and ***tlbiel*** are also used to manage the Page Walk Cache, In-Memory Table caching, and implementation-specific lookaside information that depends on the values of the PTEs. The parameters described below specify the type of translations to invalidate and the scope of the invalidation to be performed.

Radix Invalidation Control (RIC) specifies whether to invalidate the TLB, the Page Walk Cache, or both together with partition and Process Table caching. The RIC values and functions are as follows.

0 Just invalidate TLB.
1 Invalidate just Page Walk Cache.
2 Invalidate TLB, Page Walk Cache, and any caching of Partition and Process Table Entries.
3 Invalidate a group of translations (just in the TLB).

Process Scoped (PRS) specifies whether the translation(s) to be invalidated are partition scoped or process scoped including, for RIC=2, whether Process or Partition Table caching is being invalidated.

0 Invalidate partition-scoped translation(s).
1 Invalidate process-scoped translation(s).

Radix (R) specifies whether the translation(s) to be invalidated are Radix Tree translations or HPT translations. [] (R is ignored for invalidates with IS=3 and MSR_{HV}=1 because they have the potential to target translations for multiple partitions.)

0 Invalidate HPT translation(s).
1 Invalidate Radix Tree translation(s).

When MSR_{HV}=0, $LPCR_{HR}$ is used instead of R. To indicate the possibility of this substitution, the shorthand "effR" (effective R) is used in the instruction descriptions. effR=R when MSR_{HV}=1, and effR=$LPCR_{HR}$ when MSR_{HV}=0. If the effR value is incorrect for the target partition, the results of the operation are boundedly undefined.

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.

Programming Note

The hypervisor needs to be able to invalidate translations for any partition. The operating system, on the other hand, is restricted to invalidate translations for its own partition. Using effR in the description of ***tlbie*** permits this difference to be conveyed succinctly. ***tlbiel*** affects only the partition in which it is executed, so ***tlbiel*** could be described using $LPCR_{HR}$ instead of effR. effR is used for ***tlbiel*** for consistency with ***tlbie***.

The use of effR in the RTL and verbal descriptions of ***tlbie***[***l***] beginning in Version 3.1B of the architecture is a clarification of earlier architecture, not a functional change.

The IS≠0 RIC=2 variants of ***tlbie*** and ***tlbiel*** perform the same TLB invalidations as the corresponding RIC=0 variants, but in addition invalidate Page Walk Cache Entries and Partition or Process Table caching associated with the specified LPID or LPID/PID. When RIC=1 and IS≠0, the Page Walk Cache Entries for the specified LPID or LPID/PID are invalidated while leaving the corresponding TLB entries intact. The ability to target an individual Page Walk Cache Entry or the set of entries associated with a given Page Table Entry (i.e. IS=0 for RIC=1 or RIC=2) is not supported by the Power ISA. When RIC=3 and IS=0, ***tlbie*** invalidates a series of consecutive translations for HPT translation. [] For IS=0 invalidations of Radix Tree translations, the use of ***tlbie***[***l***] 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 parition 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 base virtual page number for HPT translations. The RTL, RB illustration, and verbal description for effR=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 effR=1, one for RIC=3 (requires effR=0), and one for all other cases. An implementation may choose to have a single set of encodings work consistenty between any two or all three states.

As described in Section 6.7.7, software must manage the Page Table and TLB so that there are no overlapping translations in the TLB. Failure to do so may result in ***tlbie***[***l***] specifying IS=0 not invalidating the appropriate TLB entries. []

Programming Note

As shown in the instruction descriptions, the abbreviated virtual address supplied by ***tlbie***[***l***] for HPT translations omits the high-order 14 bits of the virtual address. Translations in the TLB that differ only in these 14 bits are not considered to be overlapping translations, and a ***tlbie***[***l***] instruction with effR=0 and IS=0 that invalidates one of the corresponding TLB entries will invalidate all of them.

TLB Invalidate Entry X-form

tlbie RB,RS,RIC,PRS,R

31	RS	/	RIC	PRS	R	RB	306	/
0	6	11	12	14	15	16	21	31

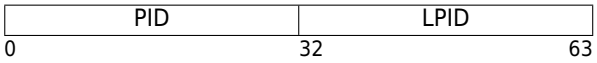
```
IS ← (RB)52:53
if MSRHV=1 then search_LPID=RS32:63
else search_LPID=LPIDRLPID
switch(IS)
  case (0b00):
    If RIC=0
      if effR=0 then
        L ← (RB)63
        if L = 0
          then inst_AVA = (RB)0:51
          else inst_AVA = (RB)0:43||(RB)56:62||0b0

        sg_size ←segment size specified in (RB)54:55
        for each thread
          for each TLB entry
            entry_scope = number of bytes of VA space translated by TLB entry
            es ← log_base_2(entry_scope)
            i = 63-es
            if (entry_VA14:i+14 = inst_AVA0:i) &
               (entry_sg_size = sg_size) &
               (entry_LPID = search_LPID) &
               (entry_process_scoped = 0)
            then TLB entry ← invalid

      else  # effR≠0
        actual_pg_size = page size specified in (RB)56:58
        p ← log_base_2(actual_pg_size)
        i = 63-p
        for each thread
          for each TLB entry
            if (entry_EA0:i = (RB)0:i) &
               (entry_actual_pg_size = actual_pg_size) &
               (entry_LPID = search_LPID) &
               (entry_process_scoped = PRS) &
               ((PRS = 0) | (entry_PID = (RS)0:31))
            then
              TLB entry ← invalid
    else if RIC=3 then
      sg_size ← segment size specified in (RB)54:55
      pg_size ← f(GS)
      number_of_pgs ← g(GS)
      p ← log_base_2(pg_size)
      n ← log_base_2(number_of_pgs)
      i ← 63-p
      va14:14+i ← (RB)0:i-n || n0
      do j=n0 to n1    # j=0 to 2n-1, in binary
        for each thread
          for each TLB entry
            entry_scope = number of bytes of VA space translated by TLB entry
            if (entry_VA14:14+i = (va14:14+i+j)) &
               (entry_sg_size = sg_size) &
               (entry_scope = pg_size) &
               (entry_LPID = search_LPID) &
               (entry_process_scoped = 0)
            then TLB entry ← invalid

  case (0b01):
    if RIC=0 | RIC=2 then
      for each TLB entry for each thread
        if (entry_LPID=search_LPID)
           & (entry_PID=RS0:31)
           & (entry_PRS=1)
```

```
        then TLB entry ← invalid
    if RIC=1 | RIC=2 then
      for each thread
        invalidate process-scoped radix page walk caching associated
        with process RS0:31 in partition search_LPID
    if (RIC=2) & (PRS=1) then
      for each thread
        invalidate Process Table caching associated with
        process RS0:31 in partition search_LPID
  case (0b10):
    if RIC=0 | RIC=2 then
      if (PRS=0) & ((MSRHV=1) | (effR=0)) then
        for each partition-scoped TLB entry for each thread
          if entry_LPID=search_LPID
          then TLB entry ← invalid
      if PRS=1 then
        for each process-scoped TLB entry for each thread
          if entry_LPID=search_LPID
          then TLB entry ← invalid
    if RIC=1 | RIC=2 then
      for each thread
        if (PRS=0)&(MSRHV=1) then
          for each thread invalidate partition-scoped page walk caching associated with
          partition search_LPID
        if PRS=1 then
          for each thread invalidate process-scoped page walk caching associated with
          partition search_LPID
    if RIC=2 then
      if (PRS=0)&(MSRHV=1) then
        for each thread invalidate Partition Table caching associated with
        partition search_LPID
      if PRS=1 then
        for each thread invalidate Process Table caching associated
        with partition search_LPID
  case (0b11):
    if RIC=0 | RIC=2 then
      if MSRHV=1 then
        for all threads
          if PRS=0 then
            all partition-scoped TLB entries ←invalid
          else
            all 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) & (effR=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 MSRHV=1 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 (MSRHV=0) & (PRS=1) then
          for each thread
            invalidate process-scoped page walk caching associated with partition search_LPID
    if RIC=2 then
      if MSRHV=1 then
        if PRS=0 then
          for each thread
            invalidate all Partition Table caching
        else
          for each thread
            invalidate all Process Table caching
      if (MSRHV=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:

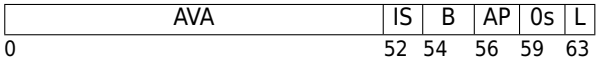


Programming Note

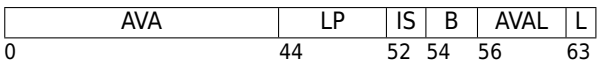
Note that although there is no PID compare for partition-scoped translation, software must still place the PID in RS when IS=0 or 1. It may be used, for example, in the TLB hash.

RB for effR=1 and IS=0b00:

RB for effR=0, IS=0b00, RIC≠3, and L=0:

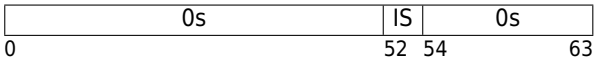


RB for effR=0, IS=0b00, RIC≠3, and L=1:



RB for effR=0, IS=0b00, and RIC=3:

RB for IS=0b01, 0b10, or 0b11:



If this instruction is executed in hypervisor state, $RS_{32:63}$ contains the search_LPID, which is the partition ID (LPID) of the partition for which one or more translations are being invalidated. Otherwise, the value in LPIDR is used as the search_LPID. 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 treated as if the instruction form were invalid.

- PRS=1, effR=0, and RIC≠2 (The only process-scoped HPT caching is of the Process Table.)
- RIC=1 and effR=0 (There is no Page Walk Cache for HPT translation.)
- RIC=3 and effR=1 (Group invalidation is only supported for HPT translation.)
- 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.)
- RIC=3 and IS≠0 (Group invalidation is only supported for individual pages.)
- PRS=0 and IS=1 (Partition-scoped translations are not associated with processes.)
- effR=0, IS=1, and RIC≠2 (HPT translations are not associated with processes.)
- effR=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.)

Architecture Note

The case described in this bullet and the corresponding bullet for ***tlbiel*** would normally be a privilege fault. Implementation constraints caused it to be treated as an invalid form. Software developers prefer not to change that in the future.

The results of an attempt to invalidate a translation outside of quadrant 0 for Radix Tree translation (effR=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.

Variable i is equal to 63 - $\log_2$ (scope of the TLB entry)

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 $RB_{56:58}$ (AP - Actual Page size field) must be set to the $SLBE_{L||LP}$ 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 $\log_2$ (actual page size specified by $(RB)_{56:58}$). The Abbreviated Virtual Address (AVA) field in register RB must contain bits 14:65 of the virtual address translated by the TLB entry to be invalidated. Variable inst_AVA is equal to the AVA field of RB.

If the L field in RB contains 1, the following rules apply.

- The base page size and actual page size are specified in the LP field in register RB, where the relationship between $(RB)_{44:51}$ (LP - Large Page size selector field) and the base page size and actual page size is the same as the relationship between PTE_{LP} and the base page size and actual page size (see Section 6.7.9.1 on page 1131 and Figure 6.14 on page 1132). Thus, b is equal to $\log_2$ (base page size specified by $(RB)_{44:51}$) and p is equal to $\log_2$ (actual page size specified by $(RB)_{44:51}$). Specifically, $(RB)_{44+c:51}$ must be equal to the contents of bits c:7 of the LP field of the PTE that was used to create the TLB entry to be invalidated, where c is the number of "r" bits in the LP field of the PTE that was used to create the TLB entry to be invalidated.
- Variable j is the larger of (63-p) and the value that is the smaller of 43 and (63-b). $(RB)_{0:j}$ must contain bits 14:(j+14) of the virtual address translated by the TLB entry to be invalidated. If b>20, $RB_{64-b:43}$ 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 entry to be invalidated, and other bits in $(RB)_{56:62}$ may contain any value and are ignored by the hardware.
- If b≥20, $(RB)_{56:62}$ (AVAL - Abbreviated Virtual Address, Lower) may contain any value and are ignored by the hardware.
- Variable inst_AVA is equal to the AVA field of RB concatenated with the AVAL field and 0b0.

Let the segment size be equal to the segment size specified in $(RB)_{54:55}$ (B field). The contents of $RB_{54:55}$ must be the same as the contents of the B field of the PTE that was used to create the TLB entry to be invalidated.

$RB_{52:53}$ and $RB_{59:62}$ (when $(RB)_{63}$ = 0) must contain zeros and are ignored by the hardware.

All TLB entries on all threads that have all of the following properties are made invalid.

- [] $VA_{14:14+i}$ is equal to $inst\_AVA_{0:i}$.
- The segment size of the entry is the same as the segment size specified in $(RB)_{54:55}$.
- []
- The entry is partition scoped.
- $TLBE_{LPID}$ = 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 $RB_{56:58}$ (AP - Actual Page size field). Thus p is equal to $\log_2$(page size specified by $RB_{56:58}$). 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 $EA_{0:i}$ 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.
- $TLBE_{LPID}$ = search_LPID.

Additional TLB entries may also be made invalid if those TLB entries contain an LPID that matches search_LPID.

If RIC=3, then the TLB entries mapping an aligned sequence of virtual pages are made invalid on all threads. The number of virtual pages in the sequence, and their page size (base page size = actual page size), are provided using an implementation-specific encoding of the GS field of RB. The number of virtual pages is a power of two. The abbreviated virtual address of the beginning of the sequence is provided by the AVA field of RB with the appropriate number of low-order bits treated as zero to cause the affected region of VA space to be aligned at a multiple of its size. The effect is as if a ***tlbie*** instruction with RIC=PRS=`eff`R=0 were executed for each virtual page in the sequence, using the supplied contents of RS and RB except using the AVA value corresponding to the virtual page and using the base and actual page size provided by GS, and with the additional match criterion that the base and actual page sizes match the scope of the TLB entry.

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 `eff`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 $TLBE_{PID}$=$RS_{0:31}$.
- 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 $RS_{0:31}$ 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 $RS_{0:31}$ 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 theads, all Process Table caching is invalidated.
- If the IS field in RB contains 0b11, MSR_{HV}=0, and PRS=1, for all threads, caching of Process Tables for the partition for which the translation is to be invalidated is invalidated.

When i>40, $RB_{40:i-1}$ 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.

Depending on the variant of the instruction, $RB_{0:51}$, $RB_{59:62}$, $RB_{59:63}$, $RB_{54:55}$, and $RB_{54:63}$ are the equivalent of reserved fields, should contain 0s, and are ignored by the hardware. When the thread is in privileged non-hypervisor state, $RS_{32:63}$ is the equivalent of a reserved field, should contain 0s, and is ignored by the hardware.

MSR_{SF} must be 1 when this instruction is executed; otherwise the results are undefined.

If the value specified in $RS_{0:31}$, $RS_{32:63}$, $RB_{54:55}$, [] $RB_{56:58}$, [] $RB_{44:51}$, or $RB_{56:63}$, 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.

The operation performed by this instruction is ordered by the ***eieio*** (or [***p***]***hwsync*** 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 $LPCR_{GTSE}$=0 or when PRS=0 and HR=1, making it hypervisor privileged.

See Section 6.10, “Translation Table Update Synchronization Requirements” for a description of other requirements associated with the use of this instruction.

Special Registers Altered:

None

Extended Mnemonics:

Extended Mnemonics for *tlbie*:

Extended mnemonic:	Equivalent to:
`tlbie RB,RS`	`tlbie RB,RS,0,0,0`

Programming Note

tlbie serves as both a basic and an extended mnemonic. The Assembler will recognize a ***tlbie*** mnemonic with five operands as the basic form, and a ***tlbie*** mnemonic with two operands as the extended form. In the extended form the RIC, PRS, and R operands are omitted and assumed to be 0.

Programming Note

Making the privilege of ***tlbie*** depend on HR instead of effR is a pragmatic choice to reduce hardware complexity. When the hypervisor executes ***tlbie***, the privilege assigned to the instruction will sometimes be wrong, but the error will not prevent the instruction from being executed.

Programming Note

In versions of the architecture that precede Version 3.1B, the actual page size was part of the match criteria for ***tlbie***[***l***], and the base page size was part of the match criteria for ***tlbie***[***l***] for HPT translations. Inclusion of the actual page size in the match criteria provided a straightforward means for software to recover from TLB multi-hit situations, but consumed extra space and made the match check more complicated, particularly for cases in which the scope of the TLB entry was less than the actual page size (possible only for HPT translations). On processors that comply with Version 3.1B of the architecture or with any subsequent version, software should use tlbie[l] specifying IS=2 for this purpose.

The actual page size continues to be provided in RB, for both HPT and Radix Tree translations, in order that it may be used by hardware to determine the scope of the TLB entries to be searched. For HPT translations the base page size continues to be supplied in RB because the distinction between base page size 4 KB and larger base page sizes is needed in order to interpret the rest of the contents of RB, and because, for base page sizes larger than 4 KB, base page size and actual page size are encoded together, in the LP field of RB.

For HPT translations, if the base page size is greater than 4 KB then, depending on the base and actual page sizes, certain bits of the virtual address may be supplied in high-order bits of the LP field of RB as well as in the AVAL field of RB. In the most extreme example, if the base and actual page sizes are both 8 KB, $LP_{0:6}$ will contain the same bits as AVAL, namely VA bits 58:64. This duplication sometimes permits hardware to obtain all needed VA bits from a contiguous sequence of bits of RB. In particular, on implementations for which the scope of TLB entries is always the actual page size, all needed VA bits can be obtained contiguously, from $RB_{0:(63-p)}$.

Programming Note

For ***tlbie***[***l***] 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***[***l***] 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***[***l***] instructions the AVA and AVAL fields in RB contain different VA bits from those in PTE_{AVA}.

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.

Architecture Note

The requirement that ***tlbie*** be executed only in 64-bit mode permits normal EA computation (in which the high-order 32 bits of the result are treated as zeros in 32-bit mode but not in 64-bit mode) to be used for ***tlbie***. (If ***tlbie*** were executed in 32-bit mode, on an implementation that does normal EA computation for ***tlbie*** the high-order 16 bits of the specified AVA bits would be treated as zeros.)

Architecture Note

The requirement that $RB_{52:53}$ and $RB_{59:62}$ (when $(RB)_{63}$ is 0), must contain zeros and be ignored by the hardware permits these bits to be assigned a meaning in some future version of the architecture.

Architecture Note

Consideration should be given to changing the definition of $RB_{44:51}$ for the ***tlbie***[***l***] instructions with $(RB)_{63}$=1 in a future version of the architecture such that $RB_{44:51}$ never contains VA bits. This would avoid redundant information in $RB_{44:51}$ and $RB_{56:62}$ and simplify the setting of $RB_{44:51}$. Alternatively, if implementations never use $RB_{56:62}$, consideration should be given to treating $RB_{56:62}$ as reserved.

Engineering Note

For process-scoped invalidations using HPT translation, performance will be better if the ERATs are not invalidated.

Engineering Note

If an ERAT contains LPID values, for IS and HV values other than IS=3 when HV=1, an ERAT entry for the specified address need be invalidated only if its LPID value matches search_LPID. If the ERAT does not contain LPID values, the ERAT entry for the specified address need be invalidated only if $LPIDR_{LPID}$ matches search_LPID. The LPIDR used for this comparison is in the same thread as the ERAT being tested.

Engineering Note

For Radix Tree translation in the ERAT, all that a partition-scoped invalidation has to match on is the LPID. So a partition invalidate in the ERAT is the only solution.

TLB Invalidate Entry Local X-form

tlbiel RB,RS,RIC,PRS,R

31	RS	/	RIC	PRS	R	RB	274	/
0	6	11	12	14	15	16	21	31

```
IS ← (RB)52:53
search_LPID=LPIDRLPID
switch(IS)
  case (0b00):
    If RIC=0
      If effR=0
        L ← (RB)63
        if L = 0 then inst_AVA = (RB)0:51
        else inst_AVA = (RB)0:43||(RB)56:62||0b0
        sg_size←segment size specified in (RB)54:55
        for each TLB entry
          entry_scope = number of bytes of VA space translated by TLB entry
          es ← log_base_2(entry_scope)
          i = 63-es
          if (entry_VA14:i+14 = inst_AVA0:i) &
             (entry_sg_size = sg_size) &
             (TLBELPID=search_LPID) &
             (entry_process_scoped = 0)
          then TLB entry ← invalid

      else  # effR≠0
        pg_size = page size specified in (RB)56:58
        p ← log_base_2(pg_size)
        i = 63-p
        for each TLB entry
          if (entry_EA0:i = (RB)0:i) &
             (entry_pg_size = pg_size) &
             (entry_LPID = search_LPID) &
             (entry_process_scoped = PRS) &
             ((PRS = 0) | (entry_PID = (RS)0:31))
          then
            TLB entry ← invalid

  case (0b01):
    if SET=0 then
      if RIC=0 | RIC=2 then
        for each TLB entry
          if (entry_LPID=search_LPID)
             & (entry_PID=RS0:31)
             & (entry_PRS=1)
          then TLB entry ← invalid
      if RIC=1 | RIC=2 then
        invalidate process-scoped radix page walk caching associated with
        process RS0:31 in partition search_LPID
      if (RIC=2)&(PRS=1) then
        invalidate Process Table caching associated with
        process RS0:31 in partition search_LPID

  case (0b10):
    if SET=0 then
      if RIC=0 | RIC=2 then
        if (PRS=0) & ((MSRHV=1) | (effR=0)) then
          for each partition-scoped TLB entry
            if entry_LPID = search_LPID
              then TLB entry ← invalid
        if PRS=1 then
          for each process-scoped TLB entry
            if entry_LPID = search_LPID
              then TLB entry ← invalid
      if RIC=1 | RIC=2 then
        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) & (MSR_HV=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 SET=0 then
      if RIC=0 | RIC=2 then
        if MSR_HV=1 then
          if PRS=0 then
            all partition-scoped TLB entries ← invalid
          else
            all process-scoped TLB entries ← invalid
        if (MSR_HV=0) & (PRS=1) then
          for each process-scoped TLB entry
            if entry_LPID=search_LPID
              then TLB entry ← invalid
        if (MSR_HV=0) & (PRS=0) & (effR=0) then
          for each partition-scoped TLB entry
            if entry_LPID=search_LPID
              then TLB entry ← invalid
      if RIC=1 | RIC=2 then
        if MSR_HV=1 then
          if PRS=0 then
            invalidate all partition-scoped page walk caching
          else
            invalidate all process-scoped page walk caching
        if (MSR_HV=0) & (PRS=1) then
          invalidate process-scoped page walk caching associated with
          partition search_LPID
      if RIC=2 then
        if MSR_HV=1 then
          if PRS=0 then
            invalidate all Partition Table caching
          else
            invalidate all Process Table caching
      if (MSR_HV=0) & (PRS=1) then
        invalidate Process Table caching associated with partition search_LPID
```

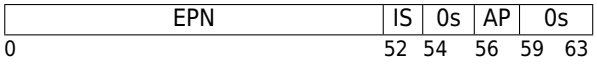
The operation performed by this instruction is based on the contents of registers RS and RB. The contents of these registers are shown below, where IS is $(RB)_{52:53}$ and L is $(RB)_{63}$.

RS:

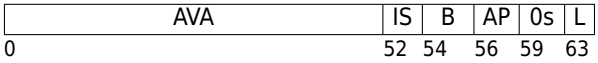
Programming Note

Note that although there is no PID compare for partition-scoped translation, software must still place the PID in RS when IS=0 or 1. It may be used, for example, in the TLB hash.

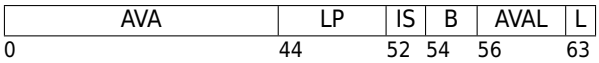
RB for effR=1 and IS=0b00:



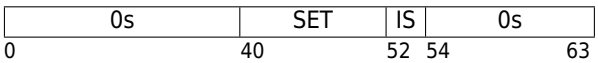
RB for effR=0, IS=0b00, and L=0:



RB for effR=0, IS=0b00, and L=1:



RB for IS=0b01, 0b10, or 0b11:



Programming Note

In versions of the architecture that precede Version 3.1, ***tlbiel*** with IS= 1, 2, or 3 invalidated appropriate entries only in a specific congruence class of the TLB, specified by SET in register RB. As a result, software was required to use a ***tlbiel*** loop to iterate through all congruence classes in order to invalidate the TLB. Software that will not be run on hardware complying with those versions should specify SET=0 in register RB. The description for ***tlbiel*** specifies SET instead of 0 in register RB to illustrate compatibility with software written to run on hardware complying with those versions.

LPIDR contains the search_LPID, which is the partition ID (LPID) of the partition for which the translation is being invalidated. $RS_{0:31}$ 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.

- []
- RIC=3 (Group invalidation is not supported for tlbiel.)

The following forms are treated as though the instruction form was invalid.

- PRS=1, `eff`R=0, and RIC≠2 (The only process-scoped HPT caching is of the Process Table.)
- RIC=1 and `eff`R=0 (There is no Page Walk Cache for HPT translation.)
- RIC=1 and IS=0 (The architecture does not support shootdown of individual translations in the Page Walk Cache.)
- RIC=1 and SET≠0 (PWC invalidation never required a loop to iterate across congruence classes.)
- RIC=2 and IS=0 (RIC is for comprehensive 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.)
- `eff`R=0, IS=1, and RIC≠2 (HPT translations are not associated with processes.)
- `eff`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 (`eff`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.

Variable i is equal to 63 - $\log_2$ (scope of the TLB entry).

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 $RB_{56:58}$ (AP - Actual Page size field) must be set to the $SLBE_{L||LP}$ 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 $\log_2$ (actual page size specified by $(RB)_{56:58}$). The Abbreviated Virtual Address (AVA) field in register RB must contain bits 14:65 of the virtual address translated by the TLB entry to be invalidated. Variable inst_AVA is equal to the AVA field of RB.

If the L field in RB contains 1, the following rules apply.

- The base page size and actual page size are specified in the LP field in register RB, where the relationship between $(RB)_{44:51}$ (LP - Large Page size selector field) and the base page size and actual page size is the same as the relationship between PTE_{LP} and the base page size and actual page size (see Section 6.7.9.1 on page 1131 and Figure 6.14 on page 1132). Thus, b is equal to $\log_2$ (base page size specified by $(RB)_{44:51}$) and p is equal to $\log_2$ (actual page size specified by $(RB)_{44:51}$). Specifically, $(RB)_{44+c:51}$ must be equal to the contents of bits c:7 of the LP field of the PTE that was used to create the TLB entry to be invalidated, where c is the number of “r” bits in the LP field of the PTE that was used to create the TLB entry to be invalidated.
- Variable i is the larger of (63-p) and the value that is the smaller of 43 and (63-b). $(RB)_{0:i}$ must contain bits 14:(i+14) of the virtual address translated by the TLB entry to be invalidated. If b>20, $RB_{64-b:43}$ 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 entry to be invalidated, and other bits in $(RB)_{56:62}$ may contain any value and are ignored by the hardware.
- If b≥20, $(RB)_{56:62}$ (AVAL - Abbreviated Virtual Address, Lower) may contain any value and are ignored by the hardware.
- Variable inst_AVA is equal to the AVA field of RB concatenated with the AVAL field and 0b0.

Let the segment size be equal to the segment size specified in $(RB)_{54:55}$ (B field). The contents of $RB_{54:55}$ must be the same as the contents of the B field of the

PTE that was used to create the TLB entry to be invalidated.

All TLB entries that have all of the following properties are made invalid on the thread executing the ***tlbiel*** instruction.

- [] $VA_{14:14+i}$ is equal to $inst\_AVA_{0:i}$.
- The segment size of the entry is the same as the segment size specified in $(RB)_{54:55}$.
- []
- The entry is partition scoped.
- $TLBE_{LPID}$ = 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 $RB_{56:58}$ (AP - Actual Page size field). Thus p is equal to $\log_2$(page size specified by $RB_{56:58}$). 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 $EA_{0:i}$ 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.
- $TLBE_{LPID}$ = search_LPID.

Additional TLB entries may also be made invalid if those TLB entries contain an LPID that matches search_LPID.

IS field in RB is non-zero

When SET=0 is specified and either RIC=0 or RIC=2, [] each partition-scoped entry when PRS=0 and either MSR_{HV}=1 or `eff`R=0, or each process-scoped entry when PRS=1 [] is invalidated if any of the following conditions are met for the entry.

- The IS field in RB contains 0b10, or MSR_{HV}=0 and the IS field contains 0b11, and $TLBE_{LPID}$ = $LPIDR_{LPID}$.
- The IS field in RB contains 0b01, $TLBE_{LPID}$=$LPIDR_{LPID}$, and $TLBE_{PID}$=$RS_{0:31}$.
- The IS field in RB contains 0b11 and MSR_{HV}=1.

When SET=0 is specified and either RIC=1 or RIC=2, if the following conditions are met, the respective partition-scoped contents when PRS=0 and MSR_{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 $LPDIR_{LPID}$ 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 $LPIDR_{LPID}$ is invalidated.

When SET=0 is specified and RIC=2, if the following conditions are met, the respective partition and Process Table caching are invalidated.

- If the IS field in RB contains 0b01 and PRS=1, caching of Process Table Entries for process $RS_{0:31}$ in partition $LPIDR_{LPID}$ is invalidated.
- If the IS field in RB contains 0b10, MSR_{HV}=1, and PRS=0, caching of Partition Tables for partition $LPIDR_{LPID}$ is invalidated.
- If the IS field in RB contains 0b10 and PRS=1, caching of Process Tables for partition $LPIDR_{LPID}$ is invalidated.
- if the IS field in RB contains 0b11, MSR_{HV}=1, and PRS=0, all Partition 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 $LIDR_{LPID}$ is invalidated.

[]

For all IS values

Any implementation specific lookaside information that is based on any TLB entry that would be invalidated by this instruction will also be invalidated.

Depending on the variant of the instruction, $RB_{0:39}$, $RB_{59:62}$, $RB_{59:63}$, $RB_{54:55}$, 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 Partition 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 $RB_{44:51}$, when it is needed to perform the specified operation, is not supported by the implementation, the instruction is treated as if the instruction form were invalid.

This instruction is privileged except when PRS=0 and HR=1, making it hypervisor privileged.

See Section 6.10, “Translation Table Update Synchronization Requirements” on page 1187 for a description of other requirements associated with the use of this instruction.

Special Registers Altered:

None

Extended Mnemonics:

Extended Mnemonics for *tlbiel*:

Extended mnemonic:	Equivalent to:
`tlbiel RB`	`tlbiel RB,0,0,0,0`

Programming Note

tlbiel serves as both a basic and an extended mnemonic. The Assembler will recognize a ***tlbiel*** mnemonic with five operands as the basic form, and a ***tlbiel*** mnemonic with one operand as the extended form. In the extended form the RS, RIC, PRS, and R operands are omitted and assumed to be 0.

Programming Note

Making the privilege of ***tlbiel*** depend on HR instead of effR is a pragmatic choice to reduce hardware complexity. When the hypervisor executes ***tlbiel***, the privilege assigned to the instruction will sometimes be wrong, but the error will not prevent the instruction from being executed.

[]

Programming Note

An operating system that uses HPT translation should only use ***tlbiel*** to invalidate the translation for a specific page when it knows whether VPM is active, and more specifically, what page size is actually in use for the target translation. The address comparison performed by ***tlbiel*** is not sensitive to whether VPM is active. As a result, the operating system must supply an AVA value that is appropriate for the page size that is in use.

Programming Note

[] See also the Programming Notes with the description of the ***tlbie*** instruction.

Architecture Note

The requirement that ***tlbiel*** be executed only in 64-bit mode permits normal EA computation (in which the high-order 32 bits of the result are treated as zeros in 32-bit mode but not in 64-bit mode) to be used for ***tlbiel***. (If ***tlbiel*** were executed in 32-bit mode, on an implementation that does normal EA computation for ***tlbiel*** the high-order 16 bits of the specified AVA bits would be treated as zeros.)

Engineering Note

If an ERAT contains LPID values, for IS and HV values other than IS=3 when HV=1, an ERAT entry for the specified address need be invalidated only if its LPID value matches search_LPID. If the ERAT does not contain LPID values, the ERAT entry for the specified address need be invalidated only if $\text{LPIDR}_{\text{LPID}}$ matches search_LPID. The LPIDR used for this comparison is in the same thread as the ERAT being tested.

TLB Synchronize X-form

tlbsync

31	///	///	///	566	/
0	6	11	16	21	31

The ***tlbsync*** instruction provides an ordering function for the effects of all ***tlbie*** instructions executed by the thread executing the ***tlbsync*** instruction, with respect to the memory barrier created by a subsequent ***ptesync*** instruction executed by the same thread. Executing a ***tlbsync*** instruction ensures that all of the following will occur.

- All TLB invalidations caused by ***tlbie*** instructions preceding the ***tlbsync*** instruction will have completed on any other thread before any data accesses caused by instructions following the ***ptesync*** instruction are performed with respect to that thread.
- All storage accesses by other threads for which the address was translated using the translations being invalidated, and all Reference and Change bit updates associated with address translations that were performed by other threads using the translations being invalidated, will have been performed with respect to the thread executing the ***ptesync*** instruction, to the extent required by the associated Memory Coherence Required attributes, before the ***ptesync*** instruction's memory barrier is created.

The operation performed by this instruction is ordered by the ***eieio*** (or [***p***]***hwsync*** or ***ptesync***) instruction with respect to preceding ***tlbie*** instructions executed by the thread executing the ***tlbsync*** instruction. The operations caused by ***tlbie*** and ***tlbsync*** are ordered by ***eieio*** as a fourth set of operations, which is independent of the other three sets that ***eieio*** orders.

The ***tlbsync*** instruction may complete before operations caused by ***tlbie*** instructions preceding the ***tlbsync*** instruction have been performed.

This instruction is privileged except when $LPCR_{GTSE}$=0, making it hypervisor privileged.

See Section 6.10 for a description of other requirements associated with the use of this instruction.

Special Registers Altered:

None

Programming Note

tlbsync should not be used to synchronize the completion of ***tlbiel***.

6.10 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 6.7.12.) Simpler optimizations are illustrated below.

The following are guidelines for safety when multiple sources of asynchronous updates are possible. To interact correctly with hardware that atomically updates Reference and Change bits (as well as with updates from other software threads), software should use atomic updates to modify valid PTEs. Academically speaking, if hardware uses simple loads and stores, software may either use locking and first invalidate the PTE and cached translations, or may attempt to optimize using atomic updates that don't change the values of the bytes containing the Reference and Change bits with the exception of potentially setting those specific bits to 1 or the Reference bit to 0. When modifying only bytes not subject to hardware modification, software may use either locking or atomic updates, subject to the limitations and optimizations described below. The realities of Reference and Change bit placement may severely limit what optimizations are possible when hardware uses normal loads and stores to update those bits.

To simplify verification and avoid paradoxes, non-impactful limitations are placed on translation table update sequence optimizations. One limitation is that software must not have two or more valid overlapping translations at any level of the translation process with different page or segment sizes. This means that one translation must be marked invalid in the translation table and invalidated from any caches prior to instating the second. The other limitation is that software must not have two or more valid translations with different attributes (i.e. WIMG, ATT). The example of I=1 and I=0 is obvious, but in general there is not enough to be gained to attempt to avoid invalidating one attribute setting before establishing another. In both of these cases, the translation cache invalidation may lag indefinitely behind the table entry invalidations and the cache invalidations may be batched, but must precede enabling the new attributes.

To protect software's ability to have reasonable performance, optimizations that hardware must support are also identified. (These optimizations are understood to be limited by the techniques used for hardware and software updates as described above, and by the properties of the table structure itself. A convention for atomic updates will yield more opportunity than locking. Hardware that does not use atomic updates may limit or eliminate the opportunity for software to optimize. The table structure for Radix Tree translation will yield more opportunity than the dual PTEG structure of HPT translation.) Access authority downgrades and setting Change bits to zero may be done without first marking the PTE invalid and invalidating the translation caches. The translation cache invalidation may lag the PTE change indefinitely and be done in bulk. Access authority upgrades and setting Reference and Change bits to 1 may be done without any PTE or translation cache invalidation. Software bits may be changed without any PTE or translation cache invalidation. Finally, any complete change to the RPN (non-overlapping with the original value) does not of itself require synchronization (though other changes to the PTE made at the same time might).

In the following examples, when the same type of sequence works for both types of translation, the HPT PTE is shown because it is more complex. In this description, and in references in subsequent subsections to "safe for multithreaded software," the safety is with respect to the risk of one thread overwriting another's update. There may also be concern for the creation of multiple matching translations, e.g. within a PTEG or pair of PTEGs. When the reservation granule is equal to or larger in size than the structure on which mutual exclusion must be ensured (e.g. PTE for Radix Tree translation but PTEG for HPT translation), multiple entries will also be prevented. (Secondary hash groups will generally not be covered by the same reservation granule as primary hash groups.)

Updates (by software) to the tables are performed only when they are known to be required by the sequential execution model (see Section 6.5). Because address translation for instructions preceding a given *Store* instruction

might cause an interrupt, and thereby prevent the corresponding store from being required by the sequential execution model, address translations for instructions preceding the *Store* instruction must be performed before the corresponding store is performed. As a result, an update to a translation table need not be preceded by a context synchronizing instruction.

All of the sequences require a context synchronizing operation after the sequence if the new contents of the translation table are to be used for address translations associated with subsequent instructions.

As noted in the description of the *Synchronize* instruction in Section 4.6.3 of Book II, address translation associated with instructions which occur in program order subsequent to the *Synchronize* (and this includes the ***ptesync*** variant) may be performed prior to the completion of the *Synchronize*. To ensure that these instructions and data which may have been speculatively fetched are discarded, a context synchronizing operation is required.

Programming Note

In many cases this context synchronization will occur naturally; for example, if the sequence is executed within an interrupt handler the ***rfid***, ***rfscv***, ***hrfid***, or ***urfid*** instruction that returns from the interrupt handler may provide the required context synchronization.

Translation table entries must not be changed in a manner that causes an implicit branch.

6.10.1 Translation Table Updates

TLBs are non-coherent caches of the HTABs and Radix Trees. TLB entries must be invalidated explicitly with one of the *TLB Invalidate* instructions. SLBs are non-coherent caches of the Segment Tables, SLB entries must be invalidated explicitly with one of the *SLB Invalidate* instructions. Page Walk Caches are non-coherent caches of the intermediate steps in Radix Tree translation. Non-coherent caching of the Partition and Process Tables is permitted. Provision has been made for the use of the *TLB Invalidate* instructions to manage the types of caching described in the preceding two sentences at a PID or LPID granularity.

Unsynchronized lookups in the Page, Segment, and when HR=0, Process Tables continue even while they are being modified. (For Partition Table Entries, and for Process Table Entries when HR=1, the process or partition affected must be inactive because the entries do not have valid bits.) With the exceptions previously identified for Segment Table walks (see Section 6.9.3, “Lookaside Buffer Management”), any thread, including a thread on which software is modifying any of the set of tables described in the first sentence, may look in those tables at any time in an attempt to translate an address. When modifying an entry in any of the former set of tables, software must ensure that the table entry’s V bit is 0 if the table entry does not correctly specify its portion of the translation (e.g., if the RPN field is not correct for the current AVA field).

For HPT translation, updates of Reference and Change bits by the hardware are not synchronized with the accesses that cause the updates. When modifying doubleword 1 of a PTE, software must take care to avoid overwriting a hardware update of these bits and to avoid having the value written by a *Store* instruction overwritten by a hardware update.

The most basic sequence that will achieve proper system synchronization for PTE updates is the following.

> ***tlbie*** instruction(s) specifying the same LPID operand value
> ***eieio***
> ***tlbsync***
> ***ptesync***

Other instructions may be interleaved among these instructions. Operating system and hypervisor software that updates Page Table Entries should use this sequence.

Operating systems and nested hypervisors are exposed to being interrupted during this sequence. The interrupting hypervisor is responsible for completing the sequence above. In general this will require the hypervisor to include the following sequence in an interrupt handler.

> ***eieio***
> ***tlbsync***
> ***ptesync***

This sequence itself may be interrupted by a higher level hypervisor. When returning to the interrupted software, the original sequence will be completed. Hardware must tolerate the result of nested interleaving of these sequences. ***tlbie*** and ***tlbsync*** instructions should only be used as part of these sequences.

The corresponding sequence for Segment Table updates uses ***slbieg*** in place of ***tlbie*** and ***slbsync*** in place of ***tlbsync***. 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 when ***tlbie*** is used for purposes other than synchronizing PTE updates, such as invalidating cached copies of Partition Table Entries or Process Table Entries (RIC=2). Mutual exclusion must be added if the update processes are multithreaded.

On systems consisting of only a single-threaded processor, the ***eieio*** and ***tlbsync*** or ***slbsync*** instructions can be omitted.

The following subsections illustrate sequences that must be used for translation table updates to tables that are subject to concurrent use by hardware (i.e. that have valid bits in their entries). For Partition Table Entries

and for Process Table Entries that do not have valid bits, simpler sequences consisting of just the preceding sequences, perhaps with mutual exclusion if the update processes are multithreaded, is sufficient.

Programming Note

The ***eieio*** instruction prevents the reordering of the preceding ***tlbie***, ***slbieg***, or ***slbiag*** instructions with respect to the subsequent ***tlbsync*** or ***slbsync*** instruction. The ***tlbsync*** or ***slbsync*** instruction and the subsequent ***ptesync*** instruction together ensure that all storage accesses for which the address was translated using the translations being invalidated (by the ***tlbie***, ***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 6.10.1.2, "Modifying a Translation Table Entry"), Reference and Change bit updates cease when the sequence is complete. When the PTE is marked invalid using an atomic update and the *Store Conditional* setting the entry invalid is successful, the Reference and Change bits obtained by the corresponding *Load And Reserve* instruction are stable/final values.

The sequences of operations shown in the following subsections assume a multi-threaded environment. In an environment consisting of only a single-threaded processor, the ***tlbsync*** or ***slbsync*** and the ***eieio*** that separates the ***tlbie*** or ***slbieg*** from the ***tlbsync*** or ***slbsync*** can be omitted. In a multi-threaded environment, when ***tlbiel*** or ***slbie*** 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. If there is no Segment Table ($LPCR_{UPRT}$=0), the synchronization requirements for using ***slbie*** apply when there is need to order SLB invalidations and/or to order storage accesses for which the address was translated using the translations being invalidated. The synchronization requirements for using ***slbia*** are the same as the synchronization requirements for using ***slbie***.

Programming Note

For all of the sequences shown in the following subsections, if it is necessary to communicate completion of the sequence to software running on another thread, the ***ptesync*** instruction at the end of the sequence should be followed by a *Store* instruction that stores a chosen value to some chosen storage location X. The memory barrier created by the ***ptesync*** instruction ensures that if a *Load* instruction executed by another thread returns the chosen value from location X, all subsequent searches of the Page or Segment Table by the other thread, that implicitly load from the PTE or STE specified by the sequence's stores, will obtain the values stored (or values stored subsequently). The *Load* instruction that returns the chosen value should be followed by a context synchronizing instruction in order to ensure that all instructions following the context synchronizing instruction will be fetched and executed using the values stored by the sequence (or values stored subsequently). (These instructions may have been fetched or executed out-of-order using the old contents of the PTE or STE.)

This Note assumes that the Page or Segment Table and location X are in storage that is Memory Coherence Required.

6.10.1.1 Adding a Page Table Entry

This is the simplest Page Table case. The V bit of the old entry is assumed to be 0. The following sequence can be used to create a PTE, maintain a consistent state, and ensure that a subsequent reference to the virtual address translated by the new entry will use the correct real address and associated attributes. A single quadword store would avoid the need for the ***eieio***. A similar sequence may be used to add a new Segment Table Entry. Mutual exclusion with respect to other software threads may be required, but there is no concern for interaction with hardware updates because the entry is invalid until the last store in the sequence.

```
PTE_pp key B ARPN LP key R C WIMG N pp ← new values
eieio          /* order 1st update before 2nd */
PTE_AVA SW L H V ← new values (V=1)
ptesync        /* order updates before next Page Table
                  search and before next data access */
```

6.10.1.2 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 (subject to the limitations described in the introduction to Section 6.10 such as avoiding overlapping translations), ensure that the translation instantiated by the old entry is no longer available, and ensure that a subsequent reference to the virtual address translated by the new entry will use the correct real

address and associated attributes.

The following sequence is to interact correctly with atomic hardware updates. It can be used to return stable Reference and Change bit values for the old translation (e.g., by setting PTE_V to 0), and is safe for multi-threaded 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 seqence (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.

```
r6 ← PTE_V L SW RPN R C ATT EAA
r4 ← addr(pte)
loop:
  ldarx r2,0,r4
  if V=0 abort, else /* to interact with locking */
  stdcx r6,0,r4
  bne-  loop
ptesync          /* order update before tlbie and
                    before next Page Table search */
tlbie(old_EA0:63-b, old_AP, old_PID, old_LPID)
eieio            /* order tlbie before tlbsync    */
tlbsync          /* order tlbie before ptesync    */
ptesync          /* complete the sequence, update
                    ordered by first ptesync      */
```

The corresponding sequence for non-atomic hardware updates is the following. (The sequence is equivalent to deleting the PTE 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.

```
PTE_V ← 0        /* (other fields don't matter) */
ptesync          /* order update before tlbie and before
                    next Page Table search      */
tlbie(old_B, old_VA14:77-b, old_L, old_LP, old_AP, old_LPID)
                 /* invalidate old translation  */
eieio            /* order tlbie before tlbsync  */
tlbsync          /* order tlbie before ptesync  */
ptesync          /* order tlbie, tlbsync and 1st
                    update before 2nd update    */
PTE_ARPN,LP,AC,R,C,WIMG,N,PP ← new values
eieio            /* order 2nd update before 3rd */
PTE_B,AVA,SW,L,H,V ← new values (V=1)
ptesync          /* order 2nd and 3rd updates
                    before next Page Table search
                    and before next data access */
```

General Case (STE)

If a valid entry is to be modified and the translation instantiated by the entry being modified is to be invalidated, the following sequence can be used to modify the STE, maintain a consistent state, ensure that the translation instantiated by the old entry is no longer available, and ensure that a subsequent reference to the effective address translated by the new entry will use the correct virtual address and associated attributes. (The sequence is much like the general case for a change to a PTE that is subject to non-atomic hardware updates, and is equivalent to deleting the STE and then adding a new one.) Mutual exclusion with respect to other software threads may be required. A similar sequence (except using ***tlbie*** with RIC=2 and ***tlbsync***) may be used to modify HR=0 Process Table Entries.

```
STE_V ← 0        /* (other fields don't matter)   */
ptesync          /* order update before slbieg and
                    before next Segment Table search */
slbieg(old_B, old_ESID, old_PID, old_LPID)
                 /* invalidate old translation    */
eieio            /* order slbieg before slbsync   */
slbsync          /* order slbieg before ptesync   */
ptesync          /* order slbieg, slbsync and 1st
                    update before 2nd update      */
    /* deletion sequence ends here */
STE_VSID, Ks, Kp, N, L, C, LP, SW ← new values
eieio            /* order 2nd update before 3rd   */
STE_ESID,V ← new values (V=1)
ptesync          /* order 2nd and 3rd updates
                    before next Segment Table search
                    and before next data access   */
```

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 overwriting another change being performed by multithreaded software, so mutual exclusion may be required.

```
oldR ← PTE_R     /* get old R                     */
if oldR = 1 then
  PTE_R ← 0      /* store byte (R=0, other
                    bits unchanged)               */
  tlbie(B,VA14:77-b,L,LP,AP,LPID)
                 /* invalidate entry              */
eieio            /* order tlbie before tlbsync    */
tlbsync          /* order tlbie before ptesync    */
ptesync          /* order tlbie, tlbsync, and update
                    before next Page Table search
                    and before next data access   */
```

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 upgrade access authority, a simpler sequence suffices because the translation hardware will refetch the PTE if an access is attempted for which the only problems were reference and/or change bits needing to be set or insufficient access authority. The store is exposed to overwriting another change being performed by multithreaded software, so mutual exclusion may be required. (Note that changing EAA_0 can be both an upgrade and a downgrade, depending on the value of Key0 of the [I]AMR. If it is not solely an upgrade, the simpler sequence must not be used.)

```
PTE_V L SW RPN R C ATT EAA ← new values (V=1)
ptesync       /* order update before next Page Table
                 search and before next data access  */
```

Modifying the SW field (PTE)

If the only change being made to a valid entry is to modify the SW field, the following sequence suffices, because the SW field is not used by the hardware (i.e. is not cached in the TLB and has no effect on hardware behavior).

```
loop:
  ldarx r1 ← PTE_dwd_0  /* load doubleword 0 of PTE */
  if V=0 abort, else    /* to interact with locking */
  r1_{57:60} ← new SW value /* replace SW, in r1      */
  stdcx. PTE_dwd_0 ← r1 /* store dwd 0 of PTE if still
                           reserved (new SW value,
                           other fields unchanged)  */
  bne- loop             /* loop if lost reservation */
```

A ***lbarx***/***stbcx.***, ***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} ← new values (V=1)
ptesync         /* order update before slbieg and
                   before next Segment Table search */
slbieg(old_B, old_ESID, old_PID, old_LPID)
                /* invalidate old translation  */
eieio           /* order slbieg before slbsync */
slbsync         /* order slbieg before ptesync */
ptesync         /* order slbieg, slbsync, and update
                   before next data access     */
```

Chapter 7. Interrupts

7.1 Overview

The Power ISA provides an interrupt mechanism to allow the thread to change state as a result of external signals, errors, or unusual conditions arising in the execution of instructions.

System Reset and Machine Check interrupts are not ordered. All other interrupts are ordered such that only one interrupt is reported, and when it is processed (taken) no program state is lost. Since Save/Restore Registers SRR0 and SRR1 are serially reusable resources used by most interrupts, program state may be lost when an unordered interrupt is taken.

7.2 Interrupt Registers

7.2.1 Machine Status Save/Restore Registers

When various interrupts occur, the state of the machine is saved in the Machine Status Save/Restore registers (SRR0 and SRR1). Section 7.5 describes which registers are altered by each interrupt.

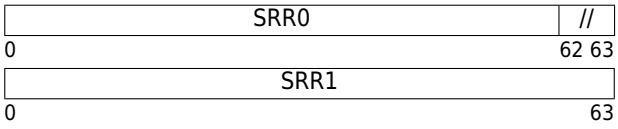


Figure 7.1: Save/Restore Registers

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 7.5.9, “Program Interrupt” on page 1215.

7.2.2 Hypervisor Machine Status Save/Restore Registers

When various interrupts occur, the state of the machine is saved in the Hypervisor Machine Status Save/Restore registers (HSRR0 and HSRR1). Section 7.5 describes which registers are altered by each interrupt.

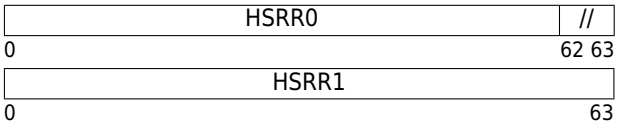


Figure 7.2: Hypervisor Save/Restore Registers

HSRR1 bits may be treated as reserved in a given implementation if they correspond to MSR bits that are reserved or are treated as reserved in that implementation and, for HSRR1 bits in the range 33:36 and 42:47, they are specified as being set either to 0 or to an undefined value for all interrupts that set HSRR1 (including implementation-dependent setting, e.g. by implementation-specific interrupts).

The HSRR0 and HSRR1 are hypervisor resources; see Chapter 2.

Programming Note

Execution of some instructions, and fetching instructions when MSR_{IR}=1 or MSR_{HV}=0, may have the side effect of modifying HSRR0 and HSRR1; see Section 7.4.4.

7.2.3 Ultravisor Machine Status Save/Restore Registers

When a Directed Ultravisor Doorbell interrupt occurs, the state of the machine is saved in the Ultravisor Machine Status Save/Restore Registers (USRR0 and USRR1).

Figure 7.3: Ultravisor Save/Restore Registers

USRR1 bits may be treated as reserved in a given implementation if they correspond to MSR bits that are reserved or are treated as reserved in that implementation and, for USRR1 bits in the range 33:36 and 42:47, they are specified as being set either to 0 or to an undefined value by the Directed Ultravisor Doorbell interrupt.

The USRR0 and USRR1 are ultravisor resources; see Chapter 3.

7.2.4 Access Segment Descriptor Register

The DAR, HDAR, SRR0, and HSRR0 generally provide the EA for storage exceptions. For hypervisor storage interrupts, additional information is often necessary to enable the hypervisor to handle the interrupt. This information is provided in a 64b SPR called the Access Segment Descriptor Register (ASDR). When nested Radix Tree translation is taking place, the ASDR will generally provide the guest real address down to bit 51. (The smallest supported page size is 4k.) When using paravirtualized HPT translation, information from the segment descriptor that was used to perform the effective to virtual translation is provided in the ASDR. For a big segment the values of the bits of the VSID field that are not part of the VSID are undefined. For exceptions that take place when translating the address of the process table entry or segment table entry group, only the VSID will be provided, because those addresses are specified as virtual addresses and the rest of the segment descriptor is implied. Some instances of the Machine Check interrupt may require the ASDR to be set similarly to how it is set for the hypervisor storage interrupts. The ASDR is set independent of the value of UPRT for the partition that is running.

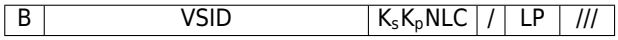


Figure 7.4: Access Segment Descriptor Register format for a Segment Descriptor

Figure 7.5: Access Segment Descriptor Register format for a Guest Real Address

7.2.5 Data Address Register

The Data Address Register (DAR) is a 64-bit register that is set by the Machine Check, Data Storage, Data Segment, and Alignment interrupts; see Sections 7.5.2, 7.5.3, 7.5.4, and 7.5.8. In general, when one of these interrupts occurs the DAR is set to an effective address associated with the storage access that caused the interrupt, with the high-order 32 bits of the DAR set to 0 if the interrupt occurs in 32-bit mode.

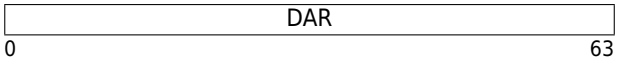


Figure 7.6: Data Address Register

7.2.6 Hypervisor Data Address Register

The Hypervisor Data Address Register (HDAR) is a 64-bit register that is set by the Hypervisor Data Storage Interrupt; see Section 7.5.16. In general, when this interrupt occurs, the HDAR is set to an effective address associated with the storage access that caused the interrupt, with the high-order 32 bits of the HDAR set to 0 if the interrupt occurs in 32-bit mode.

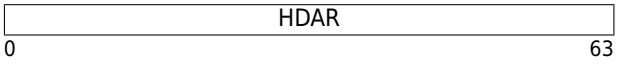


Figure 7.7: Hypervisor Data Address Register

7.2.7 Data Storage Interrupt Status Register

The Data Storage Interrupt Status Register (DSISR) is a 32-bit register that is set by the Machine Check, Data Storage, and Data Segment interrupts; see Sections 7.5.2, 7.5.3, and 7.5.4.

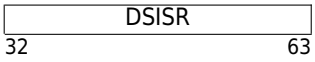


Figure 7.8: Data Storage Interrupt Status Register

DSISR bits may be treated as reserved in a given implementation if they are specified as being set either to 0 or to an undefined value for all interrupts that set the DSISR.

7.2.8 Hypervisor Data Storage Interrupt Status Register

The Hypervisor Data Storage Interrupt Status Register (HDSISR) is a 32-bit register that is set by the Hypervisor Data Storage interrupt. In general, when one of these interrupts occurs the HDSISR is set to indicate the cause of the interrupt.

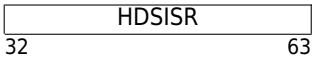


Figure 7.9: Hypervisor Data Storage Interrupt Status Register

7.2.9 Hypervisor Emulation Instruction Register

The Hypervisor Emulation Instruction Register (HEIR) is a 64-bit register that is set by the Hypervisor Emulation Assistance interrupt; see Section 7.5.18. When a word instruction causes the interrupt, the image of the instruction that caused the interrupt is loaded into bits 32:63 of the register, and bits 0:31 are set to 0s. When a prefixed instruction causes the interrupt, the image of the instruction that caused the interrupt is loaded into bits 0:63 of the register.

There may be circumstances in which the suffix cannot be loaded, such as when the instruction is located in storage that is Caching Inhibited, or when the value of the suffix corresponds to a *Branch* instruction, ***rfebb***, a context

synchronizing instruction other than ***isync***, or a "Service Processor Attention" instruction.

In such circumstances, bits 32:63 are set to 0s.

Figure 7.10: Hypervisor Emulation Instruction Register

Programming Note

When all prefixed instructions are made unavailable by the PCR setting, the prefix will be recognized as an illegal word instruction and placed in $HEIR_{32:63}$.

Architecture Note

Branch instructions, ***rfebb***, context synchronizing instructions other than ***isync***, and the "Service Processor Attention" instruction may be recoded when they are fetched into the instruction cache, before it has been determined whether they are the suffix of a prefixed instruction. If they are recoded when they appear as the suffix of a prefixed instruction that causes a Hypervisor Emulation Assistance interrupt, hardware would have difficulty loading the correct suffix into the HEIR.

7.2.10 Hypervisor Maintenance Exception Register

Each bit in the Hypervisor Maintenance Exception Register (HMER) is associated with one or more causes of the Hypervisor Maintenance exception, and is set when the associated exception(s) occur. If the corresponding bit in the Hypervisor Maintenance Exception Enable Register (HMEER) is set, a Hypervisor Maintenance Interrupt (HMI) may occur. If the thread is in a power-saving mode when the interrupt would have occurred, the thread will exit the power-saving mode; see Section 7.5.19 and Section 4.3.2.

Figure 7.11: Hypervisor Maintenance Exception Register

The contents of the HMER are as follows:

Bit	Description
0	Set to 1 for a Malfunction Alert.
1	Set to 1 when performance is degraded for thermal reasons.
2	Set to 1 when thread recovery is invoked.
Others	Implementation-specific.

When the ***mtspr*** instruction is executed with the HMER as the encoded Special Purpose Register, the contents of register RS are ANDed with the contents of the HMER and the result is placed into the HMER.

The exception bits in the HMER are sticky; that is, once set to 1 they remain set to 1 until they are set to 0 by an ***mthmer*** instruction.

Programming Note

An access to the HMER is likely to be very slow. Software should access it sparingly.

Engineering Note

The HMER must not be shared between threads. To do so would likely cause complexity for the hypervisor in support cases where both threads are handling HMIs.

Engineering Note

Some special care may be required to assure that there is no window in which a new incoming interrupt can be lost as a bit in the HMER is being reset. Hardware and software must be designed cooperatively to prevent interrupt loss. The detailed design may vary from interrupt to interrupt, and is implementation dependent.

7.2.11 Hypervisor Maintenance Exception Enable Register

The Hypervisor Maintenance Exception Enable Register (HMEER) is a 64-bit register in which each bit enables the corresponding exception in the HMER to cause the Hypervisor Maintenance interrupt, potentially causing exit from power-saving mode; see Section 7.5.19 and Section 4.3.2.

Figure 7.12: Hypervisor Maintenance Exception Enable Register

7.2.12 Facility Status and Control Register

The Facility Status and Control Register (FSCR) controls the availability of various facilities in problem state and indicates the cause of a Facility Unavailable interrupt.

When the FSCR makes a facility unavailable, attempted usage of the facility in problem state is treated as follows:

- Execution of an instruction causes a Facility Unavailable exception.
- Access of an SPR using ***mfspr***/***mtspr*** causes a Facility Unavailable exception.
- ***rfebb***, ***rfid***, ***rfscv***, ***hrfid***, ***urfid***, and ***mtmsr***[***d***] instructions have the same effect on bits in system registers as they would if the bits were available. The same is true for ***mtspr*** and ***mfspr*** unless the preceding item applies.

MMCR0 can also make various components of the Performance Monitor unavailable when accessed in problem state. An access to one of these components when it is unavailable causes a Facility Unavailable exception.

When the PCR makes a facility unavailable in problem state, the facility is treated as not defined in problem state; any Facility Unavailable interrupt that would occur if the facility were not made unavailable by the PCR does not occur.

When a Facility Unavailable interrupt occurs, the unavailable facility that was accessed is indicated in the most-significant byte of the FSCR.

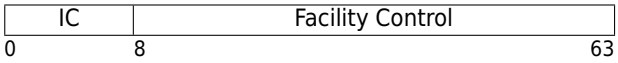


Figure 7.13: Facility Status and Control Register

The contents of the FSCR are specified below.

Bit(s) Definition

0:7 **Interruption Cause** (IC)

When a Facility Unavailable interrupt occurs, the IC field contains a binary number indicating the facility for which access was attempted. The values and their meanings are specified below.

- 02 Access to the DSCR at SPR 3
- 03 Access to a Performance Monitor SPR in group A or B when $MMCR0_{PMCC}$ is set to a value for which the access results in a Facility Unavailable interrupt. (See the definition of $MMCR0_{PMCC}$ in Section 11.4.4.)
- 04 Execution of a BHRB Instruction
- 07 Access to an Event-Based Branch SPR or execution of an Event-Based Branch instruction
- 08 Access to the Target Address Register
- 0C Execution of ***scv***
- 0D Execution of a prefixed instruction

Architecture Note

The 05 value will be among the last to be assigned a meaning. In earlier versions of the architecture it indicated that the interrupt was caused by attempt to execute a Transactional Memory instruction or to access a Transactional Memory SPR when these resources were made unavailable by a 0 value in MSR_{31}.

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.

8:49 Reserved

50 **Prefixed Instruction**

- 0 Prefixed instructions are not available in problem state.
- 1 Prefixed instructions are available in problem state unless made unavailable by another register.

51 ***scv* instruction**

- 0 The ***scv*** instruction is not available.
- 1 The ***scv*** instruction is available.

52:54 Reserved

55 **Target Address Register** (TAR)

- 0 The TAR and ***bctar*** instruction are not available in problem state.
- 1 The TAR and ***bctar*** instruction are available in problem state unless made unavailable by another register.

56 **Event-Based Branch Facility** (EBB)

- 0 The Event-Based Branch facility SPRs and instructions are not available in problem state, and event-based exceptions and branches do not occur.
- 1 The Event-Based Branch facility SPRs and instructions (see Chapter 6 of Book II) are available in problem state unless made unavailable by another register, and event-based exceptions and branches are allowed to occur if enabled by other registers.

57:60 Reserved

Programming Note

$HFSCR_{59:60}$ are used to control the availability of the Performance Monitor and the BHRB in problem and privileged non-hypervisor states. $FSCR_{59:60}$ are reserved since the availability of the Performance Monitor and BHRB is controlled by MMCR0.

61 **Data Stream Control Register at SPR 3** (DSCR)

- 0 SPR 3 is not available in problem state.
- 1 SPR 3 is available in problem state unless made unavailable by another register.

62:63 Reserved

Programming Note

When an OS has set the FSCR such that a facility is unavailable, the OS should either emulate the facility when it is accessed or provide an application interface that requires the application to request use of the facility before it accesses the facility.

7.2.13 Hypervisor Facility Status and Control Register

The Hypervisor Facility Status and Control Register (HFSCR) controls the availability of various facilities in problem and privileged non-hypervisor states, and indicates the cause of a Hypervisor Facility Unavailable interrupt.

When the HFSCR makes a facility unavailable, attempted usage of the facility in problem or privileged non-hypervisor states is treated as follows:

- Execution of an instruction causes a Hypervisor Facility Unavailable exception.
- Access of an SPR using ***mfspr***/***mtspr*** causes a Hypervisor Facility Unavailable exception
- ***rfebb***, ***rfid***, ***rfscv***, ***hrfid***, ***urfid***, and ***mtmsr***[***d***] instructions have the same effect on bits in system registers as they would if the bits were available. The same is true for ***mtspr*** and ***mfspr*** unless the preceding item applies.

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 unavailble 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.

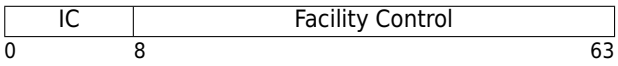


Figure 7.14: Hypervisor Facility Status and Control Register

The contents of the HFSCR are specified below.

Bit(s) Definition

0:7 **Interruption Cause** (IC)

When a Hypervisor Facility Unavailable interrupt occurs, the IC field contains a binary number indicating the access that was attempted. The values and their meanings are specified below.

- 00 Access to a Floating Point register or execution of a Floating Point instruction
- 01 Access to a Vector or VSX register or execution of a Vector or VSX instruction
- 02 Access to the DSCR at SPRs 3 or 17
- 03 Read or write access of a Performance Monitor SPR in group A, or read access of a Performance Monitor SPR in group B. (See Section 11.4.1 for a definition of groups A and B.)
- 04 Execution of a BHRB Instruction
- 07 Access to an Event-Based Branch SPR or execution of an Event-Based Branch instruction
- 08 Access to the Target Address Register
- 09 Access to the ***stop*** instruction in privileged non-hypervisor state when one or more of the following conditions exist.
  $PSSCR_{EC}$=1
  $PSSCR_{ESL}$=1
  $PSSCR_{MTL} > PSSCR_{PSLL}$
  $PSSCR_{RL} > PSSCR_{PSLL}$
- 0A Access to the ***msgsndp*** or ***msgclrp*** instructions, the TIR or the DPDES Register
- 0D Execution of a prefixed instruction

Architecture Note

The 05 value will be among the last to be assigned a meaning. In earlier versions of the architecture it indicated that the interrupt was caused by attempt to execute a Transactional Memory instruction or to access a Transactional Memory SPR when these resources were made unavailable by a 0 value in $HFSCR_{58}$.

Engineering Note

The HFSCR does not affect attempted usage of the ***stop*** instruction in problem state because ***stop*** is a privileged instruction. Any attempt to execute ***stop*** in problem state results in a Privileged Instruction type Program interrupt.

All other values are reserved.

8:63 **Facility Enable** (FE)

The FE field controls the availability of various facilities in problem and privileged non-hypervisor states as specified below.

8:49 Reserved

50 **Prefixed Instruction**

- 0 Prefixed instructions are not available in problem and privileged non-hypervisor states.
- 1 Prefixed instructions are available in problem and privileged non-hypervisor states unless made unavailable by another register.

51:52 Reserved

Programming Note

There is no bit in this register controlling the availability of the ***stop*** instruction because the availability of ***stop*** in privileged non-hypervisor state is controlled by the PSSCR. See Section 4.2.2.

53 ***msgsndp*** **instructions and SPRs** (MSGP)

- 0 The ***msgsndp*** and ***msgclrp*** instructions and the TIR and DPDES registers are not available in privileged non-hypervisor state.
- 1 The ***msgsndp*** and ***msgclrp*** instructions and the TIR and DPDES registers are available in privileged non-hypervisor state unless made unavailable by another register.

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 Reserved

Architecture Note

Bit 58 will be among the last to be assigned a meaning. It was the TM (Transactional Memory Facility) bit in earlier versions of the architecture.

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 752-754, 784-792, 795-798) are not available in privileged non-hypervisor state. (See Section 11.4.1 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 752-754, 784-792, 795-798) are available in privileged state; Performance Monitor interrupts to occur if MSR_{EE}=1 and $MMCR0_{EBE}$=0. See Section 11.2 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 MSR_{VEC} or MSR_{VSX} are not available in problem and privileged non-hypervisor states.
1 The facilities whose availability is controlled by either MSR_{VEC} or MSR_{VSX} 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 MSR_{FP} are not available in problem and privileged non-hypervisor states.
1 The facilities whose availability is controlled by MSR_{FP} are available in problem and privileged states unless made unavailable by another register.

Programming Note

The FSCR can be used to determine whether a particular facility is being used by an application, and the HFSCR can be used to determine whether a particular facility is being used by either an application or by an operating system. This is done by disabling the facility initially, and enabling it in the interrupt handler upon first usage. The information about the usage of a particular facility can be used to determine whether that facility's state must be saved and restored when changing program context.

[] Programming note moved to Section 11.4.1.

Programming Note

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).

Architecture Note

If it becomes necessary to define a facility enable bit that changes its value on interrupt, either an MSR bit will be used or two bits in the FSCR and/or HFSCR will be defined. One of the bits will enable the facility, and the other bit will be a status bit that contains the value of the enable bit when the interrupt occurred. Without this status bit, software cannot determine whether the corresponding facility was enabled prior to the interrupt if the interrupt was caused by another facility.

7.3 Interrupt Synchronization

When an interrupt occurs, SRR0, HSRR0, or USRR0 is set to point to an instruction such that all preceding instructions have completed execution, no subsequent instruction has begun execution, and the instruction addressed by SRR0, HSRR0, or USRR0 may or may not have completed execution, depending on the interrupt type.

With the exception of System Reset and Machine Check interrupts, all interrupts are context synchronizing as defined in Section 1.5.1. System Reset and Machine Check interrupts are context synchronizing if they are recoverable (i.e., if bit 62 of SRR1 is set to 1 by the interrupt). If a System Reset or Machine Check interrupt is not recoverable (i.e., if bit 62 of SRR1 is set to 0 by the interrupt), it acts like a context synchronizing operation with respect to subsequent instructions. That is, a non-recoverable System Reset or Machine Check interrupt need not satisfy items 1 through 3 of Section 1.5.1, but does satisfy items 4 and 5.

7.4 Interrupt Classes

Interrupts are classified by whether they are directly caused by the execution of an instruction or are caused by some other system exception. Those that are "system-caused" are:

- System Reset
- Machine Check
- External
- Decrementer
- Directed Privileged Doorbell
- Hypervisor Decrementer
- Hypervisor Maintenance
- Hypervisor Virtualization
- Directed Hypervisor Doorbell
- Directed Ultravisor Doorbell
- Performance Monitor

External, Decrementer, Hypervisor Decrementer, Directed Privileged Doorbell, Directed Hypervisor Doorbell, Directed Ultravisor Doorbell, Hypervisor Maintenance, and Hypervisor Virtualization interrupts are maskable interrupts. Therefore, software may delay the generation of these interrupts. System Reset and Machine Check interrupts are not maskable.

"Instruction-caused" interrupts are further divided into two classes, *precise* and *imprecise*.

7.4.1 Precise Interrupt

Except for the Imprecise Mode Floating-Point Enabled Exception type Program interrupt, all instruction-caused interrupts are precise.

When the fetching or execution of an instruction causes a precise interrupt, the following conditions exist at the interrupt point.

1. SRR0, HSRR0, and USRR0 addresses either the instruction causing the exception or the immediately following instruction. Which instruction is addressed can be determined from the interrupt type and status bits.
2. An interrupt is generated such that all instructions preceding the instruction causing the exception appear to have completed with respect to the executing thread.
3. The instruction causing the exception may appear not to have begun execution (except for causing the exception), may have been partially executed, or may have completed, depending on the interrupt type.
4. Architecturally, no subsequent instruction has begun execution.

7.4.2 Imprecise Interrupt

This architecture defines one imprecise interrupt, the Imprecise Mode Floating-Point Enabled Exception type Program interrupt.

When an Imprecise Mode Floating-Point Enabled Exception type Program interrupt occurs, the following conditions exist at the interrupt point.

1. SRR0 addresses either the instruction causing the exception or some instruction following that instruction; see Section 7.5.9, "Program Interrupt" on page 1215.
2. An interrupt is generated such that all instructions preceding the instruction addressed by SRR0 appear to have completed with respect to the executing thread.
3. The instruction addressed by SRR0 may appear not to have begun execution (except, in some cases, for causing the interrupt to occur), may have been partially executed, or may have completed; see Section 7.5.9.
4. No instruction following the instruction addressed by SRR0 appears to have begun execution.

All Floating-Point Enabled Exception type Program interrupts are maskable using the MSR bits FE0 and FE1. Although these interrupts are maskable, they differ significantly from the other maskable interrupts in that the masking of these interrupts is usually controlled by the application program, whereas the masking of all other maskable interrupts is controlled by either the operating system or the hypervisor.

7.4.3 Interrupt Processing

Associated with each kind of interrupt is an *interrupt vector*, which contains the initial sequence of instructions that is executed when the corresponding interrupt occurs.

Interrupt processing consists of saving a small part of the thread's state in certain registers, identifying the cause of the interrupt in other registers, and continuing execution at the corresponding interrupt vector location. When an exception exists that will cause an interrupt to be generated and it has been determined that the interrupt will occur, the following actions are performed. The handling of Machine Check interrupts (see Section 7.5.2) and System Call Vectored interrupts (see Section 7.5.27) differs from the description given below in several respects.

1. SRR0, HSRR0, or USRR0 is loaded with an instruction address that depends on the type of interrupt; see the specific interrupt description for details.
2. Bits 33:36 and 42:47 of SRR1, HSRR1, or USRR1 are loaded with information specific to the interrupt type.
3. Bits 0:32, 37:41, and 48:63 of SRR1, HSRR1, or USRR1 are loaded with a copy of the corresponding bits of the MSR.
4. The MSR is set as shown in Figure 7.15 on page 1202. In particular, MSR bits IR and DR are set as specified by $LPCR_{AIL}$ or $LPCR_{HAIL}$ as appropriate (see Section 2.2), and MSR bit SF is set to 1, selecting 64-bit mode. The new values take effect beginning with the first instruction executed following the interrupt.
5. Instruction fetch and execution resumes, using the new MSR value, at the effective address specific to the interrupt type. These effective addresses are shown in Figure 7.16 on page 1203. An offset may be applied to get the effective addresses, as specified by $LPCR_{AIL}$ or $LPCR_{HAIL}$ as appropriate (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.***, ***sthcx.***, ***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.***, ***sthcx.***, ***stwcx.***, ***stdcx.***, ***stqcx.***, or ***waitrsv*** on the "new" program.
- "***eieio***, ***tlbsync***, ***slbsync***, ***ptesync***," to complete any outstanding translation table modification sequence and ensure that all storage accesses caused by the interrupted program will be performed with respect to another thread before the program is resumed on that other thread. (If software conventions are such that there is no possibility of a translation table modification sequence being in progress on the thread, a [***p***]***hwsync*** instruction suffices. If software conventions are also such that there is no possibility of a ***dcbstps*** or ***dcbfps*** instruction having been executed on the thread since the preceding execution of ***phwsync*** or ***ptesync*** on the thread, an ***hwsync*** suffices.)

  If dispatching the "new" program includes changing the contents of LPIDR and/or PIDR, the ***ptesync*** or [***p***]***hwsync*** just described can, if placed properly, also serve as the ***ptesync*** or ***hwsync*** that is part of the software synchronization requirements for ***mtlpidr*** and ***mtpidr***; see note 20 in Chapter 13. If advantage is taken of this commonality, the stronger of the two required sync variants must be used. E.g., if the current Programming Note requires ***phwsync*** and note 20 in Chapter 13 requires ***hwsync***, ***phwsync*** must be used.
- ***isync*** or ***rfid*** (or ***hrfid*** or ***urfid***), to ensure that the instructions in the "new" program execute in the "new" context.
- ***cpabort***, to clear state from any previous use of the Copy-Paste Facility.

Programming Note

For instruction-caused interrupts, in some cases it may be desirable for the operating system to emulate the instruction that caused the interrupt, while in other cases it may be desirable for the operating system not to emulate the instruction. The following list, while not complete, illustrates criteria by which decisions regarding emulation should be made. The list applies to general execution environments; it does not necessarily apply to special environments such as program debugging, bring-up, etc.

In general, the instruction should be emulated if:

- The interrupt is caused by a condition for which the instruction description (including related material such as the introduction to the section describing the instruction) implies that the instruction works correctly. Example: Alignment interrupt caused by ***lmw*** for which the storage operand is not aligned, or by ***dcbz*** for which the storage operand is in storage that is Write Through Required or Caching Inhibited.
- The instruction is an illegal instruction that should appear, to the program executing it, as if it were supported by the implementation. Example: A Hypervisor Emulation Assistance interrupt is caused by an instruction that has been phased out of the architecture but is still used by some programs that the operating system supports.

If the instruction is a *Storage Access* instruction, the emulation must satisfy the atomicity requirements described in Section 1.4 of Book II.

In general, the instruction should not be emulated if:

- The purpose of the instruction is to cause an interrupt. Example: System Call interrupt caused by ***sc***.
- The interrupt is caused by a condition that is stated, in the instruction description, potentially to cause the interrupt. Example: Alignment interrupt caused by ***lwarx*** for which the storage operand is not aligned.
- The program is attempting to perform a function that it should not be permitted to perform. Example: Data Storage interrupt caused by ***lwz*** for which the storage operand is in storage that the program should not be permitted to access. (If the function is one that the program should be permitted to perform, the conditions that caused the interrupt should be corrected and the program re-dispatched such that the instruction will be re-executed. Example: Data Storage interrupt caused by ***lwz*** for which the storage operand is in storage that the program should be permitted to access but for which there currently is no PTE that satisfies the Page Table search.)

Programming Note

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.

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 $MSR_{HV\ PR\ EE}$=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, because the Directed Ultravisor Doorbell interrupt preserves MSR_{RI} instead of setting it to 0, the Directed Ultravisor Doorbell interrupt handler 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 USRRs have been restored prior to executing ***urfid***. This can be accomplished by keeping $MSR_{S\ HV\ PR\ EE}$=0b1100 during those intervals.

7.4.4 Implicit alteration of HSRR0 and HSRR1

Executing some of the more complex instructions may have the side effect of altering the contents of HSRR0 and HSRR1. The instructions listed below are guaranteed not to have this side effect. Any omission of instruction suffixes is significant; e.g., ***add*** is listed but ***add.*** is excluded.

1. *Branch* instructions
   b[***l***][***a***], ***bc***[***l***][***a***], ***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, ***urfid***
   hrfid in hypervisor state
   mtspr, ***mfspr***, ***mtmsrd***, ***mfmsr***

Programming Note

Instructions excluded from the list include the following.

- instructions that set or use XER_{CA}
- instructions that set XER_{OV} or XER_{SO}
- ***andi.***, ***andis.***, and fixed-point instructions with Rc=1 (Fixed-point instructions with Rc=1 can be replaced by the corresponding instruction with Rc=0 followed by a *Compare* instruction.)
- all floating-point instructions
- ***mftb***

These instructions, and the other excluded instructions, may be implemented with the assistance of the Hypervisor Emulation Assistance interrupt, or of implementation-specific interrupts that modify HSRR0 and HSRR1. The included instructions are guaranteed not to be implemented thus. (The included instructions are sufficiently simple as to be unlikely to need such assistance. Moreover, they are likely to be needed in interrupt handlers before HSRR0 and HSRR1 have been saved or after HSRR0 and HSRR1 have been restored.)

Similarly, fetching instructions may have the side effect of altering the contents of HSRR0 and HSRR1 unless $MSR_{HV\ IR}$ = 0b10.

7.5 Interrupt Definitions

Figure 7.15 shows all the types of interrupts and the values assigned to the MSR for each. Figure 7.16 shows the effective address of the interrupt vector for each interrupt type. (Section 6.7.5 on page 1119 summarizes all architecturally defined uses of effective addresses, including those implied by Figure 7.16.)

Interrupt Type	MSR Bit								
	IR	DR	FE0	FE1	EE	RI	ME	HV	S
System Reset	0	0	0	0	0	0	p	1	t
Machine Check	0	0	0	0	0	0	0	1	–
Data Storage	r	r	0	0	0	0	–	–	–
Data Segment	r	r	0	0	0	0	–	–	-
Instruction Storage	r	r	0	0	0	0	–	–	–
Instruction Segment	r	r	0	0	0	0	–	–	–
External	r	r	0	0	0	h	–	e	–
Alignment	r	r	0	0	0	0	–	–	–
Program	r	r	0	0	0	0	–	–	–
Floating-Point Unavailable	r	r	0	0	0	0	–	–	–
Decrementer	r	r	0	0	0	0	–	–	–
Hypervisor Decrementer	r	r	0	0	0	–	–	1	–
Directed Privileged Doorbell	r	r	0	0	0	0	–	–	–
System Call	r	r	0	0	0	0	–	s	u
Trace	r	r	0	0	0	0	–	–	–
Hypervisor Data Storage	r	r	0	0	0	–	–	1	–
Hypervisor Instruction Storage	r	r	0	0	0	–	–	1	–
Hypervisor Emulation Assistance	r	r	0	0	0	–	–	1	–
Hypervisor Maintenance	0	0	0	0	0	–	–	1	–
Directed Hypervisor Doorbell	r	r	0	0	0	–	–	1	–
Hypervisor Virtualization	r	r	0	0	0	–	–	1	–
Performance Monitor	r	r	0	0	0	0	–	–	–
Vector Unavailable	r	r	0	0	0	0	–	–	–
VSX Unavailable	r	r	0	0	0	0	–	–	–
Facility Unavailable	r	r	0	0	0	0	–	–	–
Hypervisor Facility Unavailable	r	r	0	0	0	–	–	1	–
Directed Ultravisor Doorbell	0	0	0	0	0	–	–	1	1
System Call Vectored	r	r	0	0	–	–	–	–	–

0 bit is set to 0
1 bit is set to 1
- bit is not altered
r for interrupts for which $LPCR_{AIL}$ applies, if $LPCR_{AIL}$ = [] 3, set to 1; for interrupts for which $LPCR_{HAIL}$ applies, if $LPCR_{HAIL}$=1, 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 or LEV=2, set to 1; otherwise not altered
t if the interrupt caused exit from a state-losing power-saving mode and $SMFCTRL_E$=1, set to 1; if the interrupt caused exit from a state-losing power-saving mode and $SMFCTRL_E$=0, set to 0; otherwise not altered
u if $SMFCTRL_E$ =1 and LEV=2, set to 1; otherwise not altered

Settings for Other Bits

Bits PR and PMM are set to 0.

The TE field is set to 0b00.

FP, VEC, VSX, and bits 5 and 31 are set to 0.

If the interrupt results in $MSR_{S\,HV}$ being equal to 0b11, the LE bit is copied from the UILE bit; otherwise, if the interrupt results in $MSR_{S\,HV}$ being equal to 0b01, the LE bit is copied from the HILE bit; otherwise the LE bit is copied from the $LPCR_{ILE}$ bit.

The SF bit is set to 1.

Reserved bits are set as if written as 0.

Figure 7.15: MSR setting due to interrupt

Engineering Note

If implementation-specific interrupts are going to be handled mainly by the operating system and are likely to be performance sensitive, they should be treated like the interrupts with “r” in the IR and DR columns of the table above. If the interrupt will be handled by firmware, it should be taken with IR=DR=0.

Effective Address[1]	Interrupt Type
00...0000_0100	System Reset
00...0000_0200	Machine Check
00...0000_0300	Data Storage
00...0000_0380	Data Segment
00...0000_0400	Instruction Storage
00...0000_0480	Instruction Segment
00...0000_0500	External
00...0000_0600	Alignment
00...0000_0700	Program
00...0000_0800	Floating-Point Unavailable
00...0000_0900	Decrementer
00...0000_0980	Hypervisor Decrementer
00...0000_0A00	Directed Privileged Doorbell
00...0000_0B00	Reserved
00...0000_0C00	System Call
00...0000_0D00	Trace
00...0000_0E00	Hypervisor Data Storage
00...0000_0E20	Hypervisor Instruction Storage
00...0000_0E40	Hypervisor Emulation Assistance
00...0000_0E60	Hypervisor Maintenance
00...0000_0E80	Hypervisor Doorbell
00...0000_0EA0	Hypervisor Virtualization
00...0000_0EC0	Reserved
00...0000_0EE0	Reserved for implementation-dependent interrupt for performance monitoring
00...0000_0F00	Performance Monitor
00...0000_0F20	Vector Unavailable
00...0000_0F40	VSX Unavailable
00...0000_0F60	Facility Unavailable
00...0000_0F80	Hypervisor Facility Unavailable
00...0000_0FA0	Directed Ultravisor Doorbell
00...0000_0FC0	Reserved
. . .	...
00...0000_0FFF	Reserved
00...0001_7000	System Call Vectored
00...0001_7020	System Call Vectored
. . .	...
00...0001_7FE0	System Call Vectored
00...0001_7FFF	(end of ***scv*** interrupt vectors)

[1] The values in the Effective Address column are interpreted as follows.
- 00...0000_0nnn *means* 0x0000_0000_0000_0nnn unless the values of $MSR_{S\ HV\ IR\ DR}$ and of $LPCR_{AIL}$ or $LPCR_{HAIL}$ as appropriate cause the application of an effective address offset. See the description of $LPCR_{AIL}$ and $LPCR_{HAIL}$ in Section 2.2 for more details.
- 0...00_0001_7nnn *means* 0x0000_0000_0001_7nnn unless the values of $MSR_{S\ HV\ IR\ DR}$ and of $LPCR_{AIL}$ or $LPCR_{HAIL}$ as appropriate cause the usage of an alternate effective address. See the description of $LPCR_{AIL}$ and $LPCR_{HAIL}$ 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.

Figure 7.16: Effective address of interrupt vector by interrupt type

Programming Note

When address translation is disabled, use of any of the effective addresses that are shown as reserved in Figure 7.16 risks incompatibility with future implementations.

Architecture Note

In future versions of the Architecture, interrupt vectors will be separated by a minimum of 32 bytes.

7.5.1 System Reset Interrupt

If a System Reset exception causes an interrupt that is not context synchronizing or causes the loss of a Machine Check exception or a Direct External exception, or if the state of the thread has been corrupted, the interrupt is not recoverable.

When the thread is in any power-saving level, a System Reset interrupt occurs when a System Reset exception exists. When the thread is in a power-saving level that was entered when $PSSCR_{EC}$=1, a System Reset interrupt also occurs when any of the following events occurs provided that the event is enabled to cause exit from power-saving mode (see Section 2.2). When the thread is in a power-saving level that allows the state of the LPCR to be lost, it is implementation-specific whether the following events, when enabled, cause exit, or whether only a system-reset exception causes exit.

- External
- Decrementer
- Directed Privileged Doorbell
- Directed Hypervisor Doorbell
- Directed Ultravisor Doorbell
- Hypervisor Maintenance
- Hypervisor Virtualization exception
- Implementation-specific

SRR1 indicates the exception that caused exit from power-saving mode as specified below.

The following registers are set:

SRR0 If the interrupt did not occur when the thread was in power-saving mode, set to the effective address of the instruction that the thread would have attempted to execute next if no interrupt conditions were present; if the interrupt occurred when the thread was in a power-saving mode that was entered with PSSCR bit ESL=0, and fields RL, MTL, and PSLL set to values that do not allow state loss, set to the effective address of the instruction following the stop instruction; otherwise, set to an undefined value.

If the interrupt occurred while the thread was in power-saving mode, set to the effective address of the instruction following the ***stop*** instruction when ***stop*** is executed with PSSCR bit ESL=0 and fields RL, MTL, and PSLL set to values that do not allow state loss; otherwise, set to an undefined value.

Programming Note

Whenever ***stop*** is executed in privileged non-hypervisor state, the hypervisor typically sets both $PSSCR_{ESL}$ and $PSSCR_{EC}$ 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***.

Programming Note

Since the hypervisor can ensure that no state loss occurs when ***stop*** is executed in privileged non-hypervisor state by setting PSSCR bits EC and ESL to 0s and setting field PSLL to a value that does not cause state loss (see the description of these bits in Section 4.2.2), and since the hypervisor can save the address of the instruction following the ***stop*** instruction before executing ***stop***, there is no need for the hardware to preserve the next instruction address and store it into SRR0 when state is lost. Therefore, for ease of implementation, the contents of SRR0 upon exit from a power-saving mode when state is lost are specified to be undefined.

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:

$SRR1_{42:45}$	Exception
0000	Reserved
0001	Directed Ultravisor Doorbell
0010	Implementation specific
0011	Directed Hypervisor Doorbell
0100	System Reset
0101	Directed Privlgd Doorbell
0110	Decrementer
0111	Reserved
1000	External
1001	Hypervisor Virtualization
1010	Hypervisor Maintenance
1011	Reserved
1100	Implementation specific
1101	Reserved
1110	Implementation specific
1111	Reserved

Engineering Note

Each implementation-specific exception that causes exit from a power-saving mode should use only one of the implementation-specific encodings of $SRR1_{42:45}$ to indicate the exception.

If multiple events that cause exit from power-saving mode exist, the event reported is the exception corresponding to the interrupt that would have occurred if the same conditions existed and the thread was not in power-saving mode.

46:47 Set to indicate whether the interrupt occurred when the thread was in power-saving mode and, if so, the extent to which resource state was maintained while the thread was in power-saving mode, as follows:

- **00** The interrupt did not occur when the thread was in power-saving mode.
- **01** The interrupt occurred when the thread was in power-saving mode. The state of all resources was maintained as if the thread was not in power-saving mode.
- **10** The interrupt occurred when the thread was in power-saving mode. The state of some resources was not maintained, but the state of all hypervisor resources, including the DEC, HDEC, TB, PURR, SPURR, and VTB, was maintained as if the thread was not in power-saving mode and the state of all other resources is such that the hypervisor can resume execution. (See Section 2.6 for the list of hypervisor resources.)
- **11** The interrupt occurred when the thread was in power-saving mode. The state of some resources was not maintained, and the state of some hypervisor resources was not maintained or the state of some resources is such that the hypervisor cannot resume execution.

Programming Note

Although the resources that are maintained in power-saving levels that allow loss of state are implementation-dependent, the hypervisor can avoid implementation-dependence in the portion of the System Reset and Machine Check interrupt handlers that recover from having been in power-saving mode by using the contents of $SRR1_{46:47}$, to determine what state to restore. (To avoid implementation-dependence, the hypervisor must assume that only the resources indicated in $SRR1_{46:47}$ have been preserved.)

62 If the interrupt did not occur while the thread was in a power-saving level that was entered when $PSSCR_{EC}$=1, loaded from bit 62 of the MSR if the thread is in a recoverable state; otherwise set to 0. If the interrupt occurred while the thread was in a power-saving level that was entered when $PSSCR_{EC}$=1, set to 1 if the thread is in a recoverable state; otherwise set to 0.

Engineering Note

Every attempt should be made to allow continuing execution.

Others Loaded from the MSR.

MSR See Figure 7.15 on page 1202.

In addition, if the interrupt occurs when the thread is in a power-saving level that was entered when $PSSCR_{EC}$=1 and is caused by an exception other than a System Reset exception, all other registers, except HSRR0 and HSRR1, that would be set by the corresponding interrupt if the exception occurred when the thread was not in power-saving mode are set by the System Reset interrupt, and are set to the values to which they would be set if the exception occurred when the thread was not in power-saving mode.

Execution resumes at effective address 0x0000_0000_0000_0100.

The means for software to distinguish between power-on Reset and other types of System Reset are implementation-dependent.

7.5.2 Machine Check Interrupt

The causes of Machine Check interrupts are implementation-dependent. For example, a Machine Check interrupt may be caused by a reference to a storage location that contains an uncorrectable error or does not exist (see Section 6.6), or by an error in the storage subsystem. In some cases, processor designers may choose to present what would normally be considered a storage exception, and reported as an [H]DSI or [H]ISI, as a Machine Check interrupt instead, often because of the perceived severity of the programming error or the difficulty of the verification and testing associated with reporting the error as an [H]DSI or [H]ISI. If the decision to report the exception as a Machine Check interrupt is made when the scenario is first added to the architecture, the exception may never be documented in a storage interrupt description. One such case is an attempt to access control memory as anything other than an operand of ***copy*** or ***paste.***

When the thread is not in power-saving mode, Machine Check interrupts are enabled when MSR_{ME}=1; if MSR_{ME}=0 and a Machine Check exception occurs, the thread enters the Checkstop state. When the thread is in a power-saving level that does not allow loss of hypervisor state, Machine Check interrupts are treated as enabled when $LPCR_{51}$=1 and cannot occur when $LPCR_{51}$=0. When the thread is in a power-saving level that allows loss of hypervisor state, it is implementation-specific whether Machine Check interrupts are treated as enabled $LPCR_{51}$=1 or if they cannot occur. If a Machine Check exception occurs while the thread is in power-saving mode and the Machine Check exception is not enabled to cause exit from power-saving mode, the result is implementation specific.

The Checkstop state may also be entered if an access is attempted to a storage location that does not exist (see Section 6.6), or if an implementation-dependent hardware error occurs that prevents continued operation.

Disabled Machine Check (Checkstop State)

When a thread is in Checkstop state, instruction processing is suspended and generally cannot be restarted without resetting the thread. Some implementations may preserve some or all of the internal state of the thread when entering Checkstop state, so that the state can be analyzed as an aid in problem determination.

Enabled Machine Check

If a Machine Check exception causes an interrupt that is not context synchronizing or causes the loss of a Direct External exception, or if the state of the thread has been corrupted, the interrupt is not recoverable.

The following registers are set:

SRR0 If the interrupt occurred when the thread was not in a power-saving mode, or was in a power-saving mode that was entered with PSSCR bit ESL=0, and fields RL, MTL, and PSLL set to values that do not allow state loss, set on a “best effort” basis to the effective address of some instruction that was executing or was about to be executed when the Machine Check exception occurred; otherwise set to an undefined value.

Programming Note

Since the hypervisor can ensure that no state loss occurs when ***stop*** is executed in privileged non-hypervisor state by setting PSSCR bits EC and ESL to 0s and setting field PSLL to a value that does not cause state loss (see the description of these bits in Section 4.2.2), and since the hypervisor can save the address of the instruction following the ***stop*** instruction before executing ***stop***, there is no need for the hardware to preserve the next instruction address and store it into SRR0 when state is lost. Therefore, for ease of implementation, the contents of SRR0 upon exit from a power-saving mode when state is lost are specified to be undefined.

SRR1

34 If SRR0 is set to the effective address of an instruction for which the instruction length can easily be reported, set to 0 if the instruction is a word instruction and to 1 if the instruction is a prefixed instruction; otherwise set to an undefined value.

46:47 Set to indicate whether the interrupt occurred when the thread was in power-saving mode and, if so, the extent to which resource state was maintained while the thread was in power-saving mode, as follows.

- **00** The interrupt did not occur when the thread was in power-saving mode.
- **01** The interrupt occurred when the thread was in power-saving mode. The state of all resources was maintained as if the thread was not in power-saving mode.
- **10** The interrupt occurred when the thread was in power-saving mode. The state of some resources was not maintained, but the state of all hypervisor resources, including the DEC, HDEC, TB, PURR, SPURR, and VTB, was maintained as if the thread was not in power-saving mode and the state of all other resources is such that the hypervisor can resume execution. (See Section 2.6 for the list of hypervisor resources.)
- **11** The interrupt occurred when the thread was in power-saving mode.

The state of some resources was not maintained, and the state of some hypervisor resources was not maintained or the state of some resources is such that the hypervisor cannot resume execution.

Programming Note

Although the resources that are maintained in power-saving levels that allow loss of state are implementation-dependent, the hypervisor can avoid implementation-dependence in the portion of the System Reset and Machine Check interrupt handlers that recover from having been in power-saving mode by using the contents of $SRR1_{46:47}$, to determine what state to restore. (To avoid implementation-dependence, the hypervisor must assume that only the resources indicated in $SRR1_{46:47}$ have been preserved.)

62 If the interrupt did not occur while the thread was in a power-saving level that was entered when $PSSCR_{EC}$=1, loaded from bit 62 of the MSR if the thread is in a recoverable state; otherwise set to 0. If the interrupt occurred while the thread was in a power-saving level that was entered when $PSSCR_{EC}$=1, set to 1 if the thread is in a recoverable state; otherwise set to 0.

Engineering Note

Every attempt should be made to allow continuing execution.

An error in the storage subsystem that causes a Machine Check exception does not make the resulting Machine Check interrupt non-recoverable, even if corrupted data are placed into registers, provided that an indication of the cause of the Machine Check is available to software.

Others Set to an implementation-dependent value.

MSR See Figure 7.15.

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.

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.

Engineering Note

A Machine Check interrupt caused by the existence of multiple SLB entries or TLB entries (or similar entries in implementation-specific translation caches) which translate a given effective or virtual address (see Sections 6.7.8.2 and 6.7.9.2.) must occur while still in the context of the partition that caused it. The interrupt must be presented in a way that permits continuing execution, with damage limited to the causing partition. Treating the exception as instruction-caused will achieve these requirements.

7.5.3 Data Storage Interrupt (DSI)

A Data Storage interrupt occurs when no higher priority exception exists and a data access cannot be performed for any of the reasons listed below, under the conditions described immediately below.

In general, a Data Storage interrupt can occur only under the following conditions.

- The value of the expression ($MSR_{HV\,PR}$=0b10) & (MSR_{DR}=0) is 1.
- HR=0, MSR_{DR}=1, and the value of the expression ($MSR_{HV\,PR}$=0b10) | ¬VPM is 1.
- HR=1, MSR_{DR}=1, and either of the following prevents the data access from being performed. []
  - process-scoped translation
  - partition-scoped translation for the Process Table Entry when effLPID=0

The exceptions to the preceding general statement are as follows.

- If HR=0 and $MSR_{HV\,PR}$≠0b10, a security violation in accessing the PTEG during address translation causes a Hypervisor Data Storage interrupt regardless of the rest of the translation state.
- If HR=0, MSR_{DR}=1, and an invalid Process Table Entry stops the translation process, a Data Storage interrupt occurs regardless of the rest of the translation state. (This can happen only when UPRT=1, the needed translation is not found in the SLB, and either MSR_{HV}=0 or LPID=0.)
- If HR=0, $MSR_{HV\,PR}$≠0b10, MSR_{DR}=1, VPM=0, and $LPCR_{KBV}$=1, a Virtual Page Class Key Storage Protection exception causes a Hypervisor Data Storage interrupt instead of a Data Storage interrupt, as described in more detail below.

- If HR=1, $MSR_{HV\ PR}$=0b10, and MSR_{DR}=1, an attempt to execute a *Fixed-Point Load or Store Caching Inhibited* instruction causes a Data Storage interrupt regardless of the rest of the translation state.
- If HR=1, MSR_{DR}=1, effLPID=0, and either (a) an unsupported radix tree configuration is found in the Partition Table Entry, or (b) the effective address of the Process Table Entry cannot be translated to a host real address because no valid PTE is found that translates the effective address, a Hypervisor Data Storage interrupt occurs regardless of the rest of the translation state.
- If HR=1 and either $MSR_{HV\ PR} \neq$0b10 or MSR_{DR}=1, any of the following causes a Data Storage interrupt regardless of the rest of the translation state.
  - a Data Address Watchpoint match
  - an attempt [] to execute an AMO with a reserved function code
  - an attempt to copy from control memory, or to paste to control memory that is not properly configured

Under the conditions described above, a Data Storage interrupt may occur for any of the following reasons.

- The effective or virtual address of any byte of the storage location specified by a *Load* or *Store* instruction, or by an instruction that is treated as a *Load* or *Store* other than a *Cache Block Touch* instruction, cannot be translated to a host real address (HR=0; or HR=1 and effLPID=0) or to a guest real address (HR=1 and effLPID$\neq$0) because no valid PTE was found that translates the effective or virtual address.
- HR=0 and either (a) the virtual address of the Process Table Entry or Segment Table Entry Group cannot be translated because no valid PTE was found that translates the virtual address, or (b) the Process Table Entry is invalid. []
- HR=1 and a Reference or Change bit update cannot be performed atomically in the process-scoped PTE, or in the partition-scoped PTE if effLPID=0.
- The effective address specified by a ***lq***, ***stq***, ***lwat***, ***ldat***, ***lbarx***, ***lharx***, ***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.

Programming Note

When nested Radix Tree translation is used to translate the effective address, these combinations of instruction and storage control attribute never cause a Hypervisor Data Storage interrupt (see Section 6.7.10.3).

- Control memory is specified as the source of a ***copy*** instruction or an attempt is made to paste to control memory that is not properly configured. []
- The access violates Basic Storage Protection.
- The access violates Virtual Page Class Key Storage Protection. []
- The access violates Radix Tree Translation Storage Protection. []
- The access violates Secure Memory Protection.

Programming Note

A storage protection violation may be for the data access or for an associated implicit access to a table entry, other than the PATE, during address translation or when updating Reference and Change bits. ("Table entry" here includes STEGs, PTEGs, and PDEs. For data and table entries for which the host real address is not the result of address translation, the only possible storage protection violation is a Secure Memory Protection violation.) Note that when nested Radix Tree Translation is being performed, a violation of Radix Tree Translation Storage Protection causes a DSI when the process-scoped access authority does not permit the access, regardless of whether the partition-scoped access authority permits it.

- The process- and partition-scoped page attributes (ATT values) conflict.

Programming Note

The only case of such conflict is the attribute mismatch in which the process-scoped PTE specifies I=0 and the partition-scoped PTE specifies I=1.

- An unsupported radix tree configuration is found in the Process Table Entry, in the process-scoped tables, or, if effLPID=0, in the partition-scoped tables. (Note that this condition may not be detected until the associated values are about to cause a functional problem for the processor.)
- A Data Address Watchpoint match occurs.
- An attempt is made to execute a *Load Atomic* or *Store Atomic* instruction with a reserved function code.
- An attempt is made to execute a *Fixed-Point Load or Store Caching Inhibited* instruction with MSR_{DR}=1 or specifying a storage location that is specified by the Hypervisor Real Mode Storage Control facility to be treated as non-Guarded.

Programming Note

A *Load Atomic* or *Store Atomic* instruction containing a reserved function code (see Figures 4.2 and 4.3 in Book II) is an invalid form of the instruction. Attempting to execute such a *Load Atomic* or *Store Atomic* instruction causes a DSI (or HDSI), rather than causing a Hypervisor Emulation Assistance interrupt or yielding boundedly undefined results, to permit the function code to be handled entirely by the memory agent, thereby facilitating the definition of additional function codes in the future if that proves desirable. Consistent with the usual handling of invalid instruction forms, this instance of [H]DSI occurs very early in the execution of the instruction, with the result that [hypervisor] data storage exceptions associated with computing and translating the EA do not occur and Reference and Change bits are not updated.

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:

SRR0 Set to the effective address of the instruction that caused the interrupt.

SRR1

- **33** Set to 0.
- **34** Set to 0 if the instruction that caused the interrupt is a word instruction and to 1 if the instruction that caused the interrupt is a prefixed instruction.
- **35:36** Set to 0.
- **42:47** Set to 0.
- **Others** Loaded from the MSR.

MSR See Figure 7.15.

DSISR

- **32** Set to 0.
- **33** Set to 1 if [] the translation for an attempted access is not found in the Page Table; otherwise set to 0.
- **34** Set to 1 if the process- and partition-scoped page attributes conflict; otherwise set to 0.
- **35** Set to 0.
- **36** Set to 1 if the access is not permitted by Figure 6.26 or the Privilege, Read, or Read/Write bits in Figure 6.27 as appropriate; otherwise set to 0.
- **37** Set to 1 if the access is due to a ***lq***, ***stq***, ***lwat***, ***ldat***, ***lbarx***, ***lharx***, ***lwarx***, ***ldarx***, ***lqarx***, ***stwat***, ***stdat***, ***stbcx.***, ***sthcx.***, ***stwcx.***, ***stdcx.***, or ***stqcx.*** instruction that addresses storage that is Write Through Required or Caching Inhibited; or if the access is due to a ***copy*** or ***paste.*** instruction that addresses storage that is Caching Inhibited; or if the access is due to a ***lwat***, ***ldat***, ***stwat***, or ***stdat*** instruction that addresses storage that is Guarded; otherwise set to 0.
- **38** Set to 1 for an explicit access for a *Store*, ***dcbz***, or *Load/Store Atomic* instruction; set to 1 when an implicit update of the Change bit in the process-scoped PTE fails; otherwise set to 0.
- **39:40** Set to 0.
- **41** Set to 1 if a Data Address Watchpoint match occurs; otherwise set to 0.
- **42** Set to 1 if the access is not permitted by Virtual Page Class Key Protection; otherwise set to 0.
- **43** Set to 1 if the access is not permitted by Secure Memory Protection; otherwise set to 0.
- **44** Set to 1 if an unsupported radix tree configuration is found during the translation process; otherwise set to 0.
- **45** Set to 1 if an attempt to atomically set a Reference or Change bit fails; otherwise set to 0.

Programming Note

The number of attempts hardware makes to atomically set Reference and Change bits before triggering this exception is implementation dependent. For example, the POWER9 processor makes no attempt. Software may still support the atomic update programming model to get performance benefits such as those described in Section 6.7.12.

- **46** Set to 1 if the address of the appropriate Process Table Entry or Segment Table Entry Group cannot be translated, [] or the Process Table Entry is invalid. []
- **47:59** Set to 0.
- **60** Set to 1 if control memory is specified as the source of a ***copy*** instruction [] or an attempt is made to paste to control memory that is not properly configured; [] 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 control

memory is not properly configured.

61 Set to 1 if an attempt is made to execute a *Load Atomic* or *Store Atomic* instruction specifying a reserved function code; otherwise set to 0.

62 Set to 1 if an attempt is made to execute a *Fixed-Point Load or Store Caching Inhibited* instruction with MSR_{DR}=1 or specifying a storage location that is specified by the Hypervisor Real Mode Storage Control facility to be treated as non-Guarded.

63 Set to 0.

DAR Set to the effective address of a storage element, or undefined, 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 doubleword for which access was attempted in the page that caused the exception.

- undefined, for *Load Atomic* or *Store Atomic* instruction specifying a reserved function code
- undefined, when $DSISR_{60}$=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 doubleword for which access was attempted in the page that caused the exception, for a *Load* or *Store* instruction or an instruction that is treated as a *Load* or *Store* ("first" refers to address order: see Section 7.7).
- the first byte of overlap between the operand and the matching watched range, for a Data Address Watchpoint match

For the cases in which the DAR is specified above to be set to a defined value, if the interrupt occurs in 32-bit mode the high-order 32 bits of the DAR are set to 0.

If a Data Storage exception occurs due to an attempt to execute a *Fixed-Point Load or Store Caching Inhibited instruction* when MSR_{DR}=1 or a *Load Atomic* or *Store Atomic* instruction with a reserved function code, the only exception bit that is set in the DSISR is bit 62 or 61 respectively and the remainder of this paragraph does not apply. Otherwise, if multiple Data Storage exceptions occur for a given effective address, any one or more of the bits corresponding to these exceptions may be set to 1 in the DSISR. However, if one or more Data Storage exceptions are to be reported together with a Virtual Page Class Key Storage Protection exception that occurs when the thread is not in hypervisor state ($MSR_{HV\ PR}$≠0b10), $LPCR_{KBV}$=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 7.16.

Architecture Note

In general, if an instruction that accesses data storage causes an interrupt, and the attempted access should be emulated by the interrupt handler, the interrupt should be an Alignment interrupt rather than a [Hypervisor] Data Storage interrupt.

Engineering Note

For initial hardware debug it is often useful to run with cache disabled. In some ways cache disabled mode is similar to Caching Inhibited storage. Although ***lq*** and ***stq*** need not be supported by an implementation for storage that is Caching Inhibited, support for ***lq*** and ***stq*** with cache disabled should be considered.

7.5.4 Data Segment Interrupt

For Paravirtualized HPT Translation, a Data Segment interrupt occurs when no higher priority exception exists and a data access cannot be performed because data address translation is enabled and the effective address of any byte of the storage location specified by a *Load* or *Store* instruction, or by an instruction that is treated as a *Load* or *Store* other than a *Cache Block Touch* instruction, cannot be translated to a virtual address because the SLB search fails and, if UPRT=1, the Segment Table search fails after the STEG has been accessed.

For Radix Tree Translation, a Data Segment interrupt occurs when no higher priority exception exists and a data access cannot be performed because for the effective address specified by a *Load Store* [] instruction, or by an instruction that is treated as a *Load* or *Store* other than a *Cache Block Touch* instruction, one of the following conditions is satisfied.

- Data address translation is enabled and $EA_{0:1}$=0b01 or 0b10 when $MSR_{HV\ PR} \neq$ 0b10.
- Data address translation is enabled and $EA_{0:1}$=0b11 when MSR_{PR}=1.
- Data address translation is enabled and $EA_{0:1}$=0b00 when $MSR_{HV\ PR}$=0b10 and LPIDR≠0.
- $EA_{2:63}$ 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:

SRR0 Set to the effective address of the instruction that caused the interrupt.

SRR1

33 Set to 0.

34 Set to 0 if the instruction that caused the interrupt is a word instruction and to 1 if the instruction that caused the interrupt is a prefixed instruction.

35:36 Set to 0.

42:47 Set to 0.

Others Loaded from the MSR.

MSR See Figure 7.15.

DSISR Set to an undefined value.

DAR Set to the effective address of a storage element as described in the following list.

- a byte in the block that caused the exception, for a *Cache Management* instruction
- a byte in the first aligned doubleword for which access was attempted in the segment that caused the exception, for a *Load* or *Store* instruction, or for an instruction that is treated as a *Load* or *Store* other than a *Cache Management* instruction, ("first" refers to address order: see Section 7.7).

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_0380, possibly offset as specified in Figure 7.16.

Programming Note

A Data Segment interrupt occurs if MSR_{DR}=1 and the translation of the effective address of any byte of the specified storage location is not found in the SLB (or in any implementation-specific address translation lookaside information).

7.5.5 Instruction Storage Interrupt (ISI)

An Instruction Storage interrupt occurs when no higher priority exception exists and the next instruction to be executed cannot be fetched for any of the reasons listed below, under the conditions described immediately below.

In general, an Instruction Storage interrupt can occur only under the following conditions.

- The value of the expression ($MSR_{HV\ PR}$=0b10) & (MSR_{IR}=0) is 1.
- HR=0, MSR_{IR}=1, and the value of the expression ($MSR_{HV\ PR}$=0b10) | ¬VPM is 1. []
- HR=1, MSR_{IR}=1, and either of the following prevents the next instruction to be executed from being fetched. []
  - process-scoped translation
  - partition-scoped translation for the Process Table Entry when effLPID=0

The exceptions to the preceding general statement are as follows.

- If HR=0 and $MSR_{HV\ PR}$≠0b10, a security violation in accessing the PTEG during address translation causes a Hypervisor Instruction Storage interrupt regardless of the rest of the translation state.
- If HR=0, MSR_{IR}=1, and an invalid Process Table Entry stops the translation process, an Instruction Storage interrupt occurs regardless of the rest of the translation state. (This can happen only when UPRT=1, the needed translation is not found in the SLB, and either MSR_{HV}=0 or LPID=0.)
- If HR=1, MSR_{IR}=1, effLPID=0, and either (a) an unsupported radix tree configuration is found in the Partition Table Entry, or (b) the effective address of the Process Table Entry cannot be translated to a host real address because no valid PTE is found that translates the effective address, a Hypervisor Instruction Storage interrupt occurs regardless of the rest of the translation state.
- If the instruction is a prefixed instruction and is in storage that is Caching Inhibited, attempting to fetch it causes an Instruction Storage interrupt regardless of the rest of the translation state.

Under the conditions described above, an Instruction Storage interrupt may occur for any of the following reasons.

- The effective or virtual address of the next instruction to be executed cannot be translated to a host real address (HR=0; or HR=1 and effLPID=0) or to a guest real address (HR=1 and effLPID≠0) because no valid PTE was found that translates the effective or virtual address.
- HR=0 and either (a) the virtual address of the Process Table Entry or Segment Table Entry Group cannot be translated because no valid PTE was found that translates the virtual address, or (b) the Process Table Entry is invalid. []
- HR=1 and a Reference bit update cannot be performed atomically in the process-scoped PTE, or in the partition-scoped PTE if effLPID=0.
- The instruction is a prefixed instruction and is in storage that is Caching Inhibited.
- The access violates Basic Storage Protection.
- The access violates Virtual Page Class Key Storage Protection.

- The access violates Radix Tree Translation Storage Protection.
- The access violates Secure Memory Protection.

Programming Note

A storage protection violation may be for the instruction fetch or for an associated implicit access to a table entry, other than the PATE, during address translation or when updating Reference bits. ("Table entry" here includes STEGs, PTEGs, and PDEs. For instructions and table entries for which the host real address is not the result of address translation, the only possible storage protection violation is a Secure Memory Protection violation.) Note that when nested Radix Tree Translation is being performed, a violation of Radix Tree Translation Storage Protection causes an ISI when the process-scoped access authority does not permit the access, regardless of whether the partition-scoped access authority permits it.

- The process- and partition-scoped page attributes (ATT values) conflict.

Programming Note

The only case of such conflict is the attribute mismatch in which the process-scoped PTE specifies I=0 and the partition-scoped PTE specifies I=1.

- An unsupported radix tree configuration is found in the Process Table Entry, in the process-scoped tables, or, if effLPID=0, in the partition-scoped tables. (Note that this condition may not be detected until the associated values are about to cause a functional problem for the processor.)

The following registers are set:

SRR0 Set to the effective address of the instruction that the thread would have attempted to execute next if no interrupt conditions were present (if the interrupt occurs on attempting to fetch a branch target, SRR0 is set to the branch target address).

SRR1

33 Set to 1 if [] the translation for an attempted access is not found in the Page Table; otherwise set to 0.

34 Set to 1 if the process- and partition-scoped page attributes conflict; otherwise set to 0.

35 Set to 1 if the access is to No-execute (as indicated by the N bit in the Segment Table Entry or the N bit in the HPT PTE or the Execute and Privilege bits in the EAA field of the Radix PTE and IAMR key 0) or Guarded storage, or is to Caching Inhibited storage and is for a prefixed instruction; otherwise set to 0.

36 Set to 1 if the access is not permitted by Figure 6.26 or the Privilege or Execute bits in Figure 6.27 as appropriate; otherwise set to 0.

42 Set to 1 if the access is not permitted by Virtual Page Class Key Protection; otherwise set to 0.

43 Set to 1 if the access is not permitted by Secure Memory Protection; otherwise set to 0.

44 Set to 1 if an unsupported radix tree configuration is found during the translation process; otherwise set to 0.

45 Set to 1 if an attempt to atomically set a Reference 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. For example, the POWER9 processor makes no attempt. Software may still support the atomic update programming model to get performance benefits such as those described in Section 6.7.12.

46 Set to 1 if the address of the appropriate Process Table Entry or Segment Table Entry Group cannot be translated, [] or the Process Table Entry is invalid. []

47 Set to 0.

Others Loaded from the MSR.

MSR See Figure 7.15.

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 7.16.

7.5.6 Instruction Segment Interrupt

For Paravirtualized HPT Translation, an Instruction Segment interrupt occurs when no higher priority exception exists and the next instruction to be executed cannot be fetched because instruction address translation is enabled and the effective address cannot be translated to a virtual address because the SLB search fails and, if UPRT=1, the Segment Table search fails after the STEG has been accessed.

For Radix Tree Translation, an Instruction Segment interrupt occurs when no higher priority exception exists and the next instruction to be executed cannot be fetched because one of the following conditions is satisfied.

- Instruction address translation is enabled and $EA_{0:1}$=0b01 or 0b10 when $MSR_{HV\ PR} \neq$ 0b10.
- Instruction address translation is enabled and $EA_{0:1}$=0b11 when MSR_{PR}=1.
- Instruction address translation is enabled and $EA_{0:1}$=0b00 when $MSR_{HV\ PR}$=0b10 and LPIDR≠0.
- $EA_{2:63}$ is outside the range translated by the appropriate Radix Tree.

The following registers are set:

SRR0 Set to the effective address of the instruction that the thread would have attempted to execute next if no interrupt conditions were present (if the interrupt occurs on attempting to fetch a branch target, SRR0 is set to the branch target address).

SRR1

33:36 Set to 0.

42:47 Set to 0.

Others Loaded from the MSR.

MSR See Figure 7.15 on page 1202.

Execution resumes at effective address 0x0000_0000_0000_0480, possibly offset as specified in Figure 7.16.

Programming Note

An Instruction Segment interrupt occurs if MSR_{IR}=1 and the translation of the effective address of the next instruction to be executed is not found in the SLB (or in any implementation-specific address translation lookaside information).

7.5.7 External Interrupt

An External interrupt is classified as being either a Direct External interrupt or a Mediated External interrupt. Throughout this Book, usage of the phrase “External interrupt’, without further classification, refers to both a Direct External interrupt and a Mediated External interrupt.

7.5.7.1 Direct External Interrupt

A Direct External interrupt occurs when no higher priority exception exists, a Direct External exception exists, and the value of the expression

MSR_{EE} & ¬(MSR_{HV} & ¬MSR_{PR} & $LPCR_{HEIC}$) | (¬(LPES) & (¬(MSR_{HV}) | MSR_{PR}))

is one. The occurrence of the interrupt does not cause the exception to cease to exist.

Programming Note

When HEIC=1, Direct External exceptions will not result in external interrupts when the processor is in hypervisor state even if MSR_{EE}=1. This enables the Hypervisor Virtualization interrupt handler to prevent External interrupts from occurring during the Hypervisor Virtualization interrupt handler.

When LPES=0, the following registers are set:

HSRR0 Set to the effective address of the instruction that the thread would have attempted to execute next if no interrupt conditions were present.

HSRR1

33:36 Set to 0.

42:47 Set to 0.

Others Loaded from the MSR.

MSR See Figure 7.15 on page 1202.

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 7.15 on page 1202.

Execution resumes at effective address 0x0000_0000_0000_0500, possibly offset as specified in Figure 7.16.

Programming Note

Because the value of MSR_{EE} is always 1 when the thread is in problem state, the simpler expression

MSR_{EE} & ¬(MSR_{HV} & ¬MSR_{PR} & $LPCR_{HEIC}$) | ¬(LPES | MSR_{HV})

is equivalent to the expression given above.

Programming Note

The Direct External exception has the same meaning as the External exception in versions of the architecture prior to Version 2.05.

7.5.7.2 Mediated External Interrupt

A Mediated External interrupt occurs when no higher priority exception exists, a Mediated External exception exists (see the definition of $LPCR_{MER}$ in Section 2.2), and the value of the expression

MSR_{EE} & (¬(MSR_{HV}) | MSR_{PR})

is one. The occurrence of the interrupt does not cause the exception to cease to exist.

When LPES=0, the following registers are set:

HSRR0 Set to the effective address of the instruction that the thread would have attempted to execute next if no interrupt conditions were present.

HSRR1

33:36 Set to 0.

42 Set to 1.

43:47 Set to 0.

Others Loaded from the MSR.

MSR See Figure 7.15 on page 1202.

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 7.15 on page 1202.

Execution resumes at effective address 0x0000_0000_0000_0500, possibly offset as specified in Figure 7.16.

7.5.8 Alignment Interrupt

Most causes of Alignment interrupt involve storage operands, and many of those causes involve the alignment thereof. Storage operand alignment is defined in Section 1.10.1 of Book I. Another cause of Alignment interrupt is attempt to execute a prefixed instruction that crosses a 64-byte address boundary.

An Alignment interrupt occurs when no higher priority exception exists and an attempt is made to execute an instruction in a manner that is required, by the instruction description, to cause an Alignment interrupt, or to execute a prefixed instruction that crosses a 64-byte boundary. These cases are as follows.

- A *Load/Store Multiple* instruction that is executed in Little-Endian mode
- A *Move Assist* instruction that is executed in Little-Endian mode, unless the string length is zero
- A ***copy***, ***paste.***, ***lwat***, ***ldat***, ***lharx***, ***lwarx***, ***ldarx***, ***lqarx***, ***stwat***, ***stdat***, ***sthcx.***, ***stwcx.***, ***stdcx.***, or ***stqcx.***, ***hashst***, ***hashchk***, ***hashstp*** or ***hashchkp*** instruction that has an unaligned storage operand, unless execution of the instruction yields boundedly undefined results
- The operand(s) of a *Load Atomic* or *Store Atomic* instruction cross(es) a 32-byte boundary.
- A prefixed instruction that is at an effective address equal to 60 modulo 64.

An Alignment interrupt may occur when no higher priority exception exists and a data access cannot be performed for any of the following reasons.

- The storage operand of ***lfdp***, ***lfdpx***, ***stfdp***, ***stfdpx***, ***lxsihzx***, or ***stxsihx*** is unaligned.
- The storage operand of ***lq*** or ***stq*** is unaligned.
- The storage operand of a Floating-Point *Storage Access* or VSX *Storage Access* instruction other than ***lfdp***, ***lfdpx***, ***stfdp***, ***stfdpx***, ***lxsihzx***, ***lxsibzx***, ***stxsihx***, or ***stxsibx*** is not word-aligned.
- The storage operand of a *Load/Store Multiple* [] instruction is not [] aligned and the thread is in Big-Endian mode.
- The storage operand of a *Load/Store Multiple*, ***lfdp***, ***lfdpx***, ***lxvl***, ***lxvll***, ***stfdp***, ***stfdpx***, ***stxvl***, ***stxvll***, 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 Set to 0.

34 Set to 0 if the instruction that caused the interrupt is a word instruction and to 1 if the instruction that caused the interrupt is a prefixed instruction.

35 Set to 1 if the instruction that caused the interrupt is a prefixed instruction that is at an effective address equal to 60 modulo 64; otherwise set to 0.

36 Set to 0.

42:47 Set to 0.

Others Loaded from the MSR.

MSR See Figure 7.15.

DAR Set to the effective address computed by the instruction, unless the instruction is a prefixed instruction at an effective address equal to 60 modulo 64, in which case set to an undefined value except as described in the next sentence. If the interrupt occurs in 32-bit mode the high-order 32 bits of the DAR are set to 0.

Execution resumes at effective address 0x0000_0000_0000_0600, possibly offset as specified in Figure 7.16.

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.

Engineering Note

It is recommended that all data accesses described in the second bulleted list above be performed in hardware (unless they cause another kind of interrupt) rather than cause an Alignment interrupt.

7.5.9 Program Interrupt

A Program interrupt occurs when no higher priority exception exists and one of the following exceptions arises during execution of an instruction:

Floating-Point Enabled Exception

A Floating-Point Enabled Exception type Program interrupt is generated when the value of the expression

$$(MSR_{FE0} \mid MSR_{FE1}) \ \& \ FPSCR_{FEX}$$

is 1. $FPSCR_{FEX}$ is set to 1 by the execution of a floating-point instruction that causes an enabled exception, including the case of a *Move To FPSCR* instruction that causes an exception bit and the corresponding enable bit both to be 1.

Privileged Instruction

The following applies if the instruction is executed when MSR_{PR} = 1.

A Privileged Instruction type Program interrupt is generated when execution is attempted of a privileged instruction, or of an ***mtspr*** or ***mfspr*** instruction with an SPR field that contains a value having spr_0=1.

The following applies if the instruction is executed when $MSR_{HV\ PR}$ = 0b00 and $LPCR_{EVIRT}$=0.

A Privileged Instruction type Program interrupt is generated when 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 or that specifies PTCR, DAWRn, DAWRXn, or CIABR when those SPRs are ultravisor privileged for the operation.

The following applies if the instruction is executed when $MSR_{HV\ PR}$ = 0b00 or when $MSR_{S\ HV\ PR}$ = 0b010.

A Privileged Instruction type Program interrupt is generated when execution is attempted of an ultravisor privileged instruction, or of an ***mtspr*** or ***mfspr*** instruction that specifies an SPR, other than PTCR, DAWRn, DAWRXn, and CIABR, that is ultravisor privileged for the operation.

Programming Note

These are the only cases in which a Privileged Instruction type Program interrupt can be generated when MSR_{PR}=0. They can be distinguished from other causes of Privileged Instruction type Program interrupts by examining $SRR1_{49}$ (the bit in which MSR_{PR} was saved by the interrupt).

Trap

A Trap type Program interrupt is generated when any of the conditions specified in a *Trap* instruction is met, or when the two values compared by a ***hashchk*** or ***hashchkp*** instruction are unequal.

The following registers are set:

SRR0 For all Program interrupts except a Floating-Point Enabled Exception type Program interrupt, set to the effective address of the instruction that caused the corresponding exception.

For a Floating-Point Enabled Exception type Program interrupt, set as described in the following list.

- If $MSR_{FE0\ FE1}$ = 0b00, $FPSCR_{FEX}$ = 1, and an instruction is executed that changes $MSR_{FE0\ FE1}$ to a nonzero value, set to the effective address of the instruction that the thread would have attempted to execute next if no interrupt conditions were present.

  Programming Note

  Recall that all instructions that can alter $MSR_{FE0\ FE1}$ are context synchronizing, and therefore are not initiated until all preceding instructions have reported all exceptions they will cause.

- If $MSR_{FE0\ FE1}$ = 0b11, set to the effective address of the instruction that caused the Floating-Point Enabled Exception.
- If $MSR_{FE0\ FE1}$ = 0b01 or 0b10, set to the effective address of the first instruction that caused a Floating-Point Enabled Exception since the most recent time $FPSCR_{FEX}$ 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

33 Set to 0.

34 Except for the cases in which SRR0 is not necessarily set to the effective address of the instruction that caused the exception (first and third bullets in the description of how SRR0 is set), set to 0 if the instruction that caused the interrupt is a word instruction and to 1 if the instruction that caused the interrupt is a prefixed instruction; for the cases in which SRR0 is not necessarily set to the effective address of the instruction that caused the exception, set to an undefined value.

35:36 Set to 0.

42 Set to 0.

Architecture Note

Bit 42 will be among the last to be assigned a meaning. In earlier versions of the architecture it indicated that a "TM Bad Thing type Program interrupt" had occurred.

43 Set to 1 for a Floating-Point Enabled Exception type Program interrupt; otherwise set to 0.

44 Set to 0.

Programming Note

Bit 44 will not be assigned a meaning. In versions of the architecture that precede Version 2.05 this bit was set to 1 (and bits 42:43 and 45:46 were set to 0) to indicate that an "Illegal Instruction type Program interrupt" had occurred. Hypervisors may set this bit to 1 as part of simulating an Illegal Instruction type Program interrupt to the operating system, as described in a subsequent Programming Note.

45 Set to 1 for a Privileged Instruction type Program interrupt; otherwise set to 0.

46 Set to 1 for a Trap type Program interrupt; otherwise set to 0.

47 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.

Programming Note

$SRR1_{47}$ can be set to 1 only if the exception is a Floating-Point Enabled Exception and either $MSR_{FE0\ FE1}$ = 0b01 or 0b10 or $MSR_{FE0\ FE1}$ has just been changed from 0b00 to a nonzero value. ($SRR1_{47}$ is always set to 1 in the last case.)

Others Loaded from the MSR.

Exactly one of bits 43, 45, and 46 is set to 1.

MSR See Figure 7.15 on page 1202.

Execution resumes at effective address 0x0000_0000_0000_0700, possibly offset as specified in Figure 7.16.

Programming Note

In versions of the architecture that precede V. 2.05, the conditions that now cause a Hypervisor Emulation Assistance interrupt with $HSRR1_{45}$=0 instead caused an "Illegal Instruction type Program interrupt". This was a Program interrupt for which registers (SRR0, SRR1, and the MSR) were set as described above for the Privileged Instruction type Program interrupt, except that $SRR1_{44}$ was set to 1 and $SRR1_{45}$ was set to 0. Thus older operating systems have code to handle these conditions, at the Program interrupt vector location. For this reason, if a Hypervisor Emulation Assistance interrupt occurs with $HSRR1_{45}$=0 when the thread is not in hypervisor state, for an instruction that the hypervisor determines should be handled by the operating system, the hypervisor is expected to pass control to the operating system at the operating system's Program interrupt vector location, with all registers (SRR0, SRR1, MSR, GPRs, etc.) set as if the instruction had caused a Privileged Instruction type Program interrupt, except with $SRR1_{44:45}$ set to 0b10. (The Hypervisor Emulation Assistance interrupt was added to the architecture in V. 2.05, and the Illegal Instruction type Program interrupt was removed from the architecture in V. 2.06. In V. 2.05 the Hypervisor Emulation Assistance interrupt was optional: implementations that supported it generated it as described in V. 2.06, and never generated an Illegal Instruction type Program interrupt; implementations that did not support it generated an Illegal Instruction type Program interrupt as described above.)

Programming Note

When $LPCR_{EVIRT}$=1, some of the conditions that cause a Privileged Instruction type Program interrupt when $LPCR_{EVIRT}$=0 (attempted execution, in privileged but non-hypervisor state, of a hypervisor privileged instruction or of an ***mtspr*** or ***mfspr*** instruction specifying an SPR that is hypervisor privileged for the operation or PTCR, DAWRn, DAWRXn, or CIABR when they are ultravisor privileged for the operation) instead cause a Hypervisor Emulation Assistance interrupt with $HSRR1_{45}$=1. Having these conditions cause a Hypervisor Emulation Assistance interrupt permits support of nested hypervisors through virtualization of hypervisor resources, and simplifies creation of a common kernel for the OS and the hypervisor. In versions of the architecture that precede V. 3.0, $LPCR_{EVIRT}$ did not exist and these conditions always caused a Privileged Instruction type Program interrupt. Thus older operating systems have code to handle these conditions, at the Program interrupt vector location. For this reason, if a Hypervisor Emulation Assistance interrupt occurs with $HSRR1_{45}$=1 for an instruction that the hypervisor determines should be handled by the operating system, the hypervisor is expected to pass control to the operating system at the operating system's Program interrupt vector location, with all registers (SRR0, SRR1, MSR, GPRs, etc.) set as if the instruction had caused a Privileged Instruction type Program interrupt.

7.5.10 Floating-Point Unavailable Interrupt

A Floating-Point Unavailable interrupt occurs when no higher priority exception exists, an attempt is made to execute a floating-point instruction (including floating-point loads, stores, and moves), and MSR_{FP}=0.

The following registers are set:

SRR0 Set to the effective address of the instruction that caused the interrupt.

SRR1

- **33** Set to 0.
- **34** Set to 0 if the instruction that caused the interrupt is a word instruction and to 1 if the instruction that caused the interrupt is a prefixed instruction.
- **35:36** Set to 0.
- **42:47** Set to 0.
- **Others** Loaded from the MSR.

MSR See Figure 7.15 on page 1202.

Execution resumes at effective address 0x0000_0000_0000_0800, possibly offset as specified in Figure 7.16.

7.5.11 Decrementer Interrupt

A Decrementer interrupt occurs when no higher priority exception exists, a Decrementer exception exists, and MSR_{EE}=1.

The following registers are set:

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 7.15 on page 1202.

Execution resumes at effective address 0x0000_0000_0000_0900, possibly offset as specified in Figure 7.16.

7.5.12 Hypervisor Decrementer Interrupt

A Hypervisor Decrementer interrupt occurs when no higher priority exception exists, a Hypervisor Decrementer exception exists, and the value of the following expression is 1.

$(MSR_{EE} \mid \neg(MSR_{HV}) \mid MSR_{PR})$ & 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 7.15 on page 1202.

Execution resumes at effective address 0x0000_0000_0000_0980, possibly offset as specified in Figure 7.16.

Programming Note

Because the value of MSR_{EE} is always 1 when the thread is in problem state, the simpler expression

$(MSR_{EE} \mid \neg(MSR_{HV}))$ & HDICE

is equivalent to the expression given above.

7.5.13 Directed Privileged Doorbell Interrupt

A Directed Privileged Doorbell interrupt occurs when no higher priority exception exists, a Directed Privileged Doorbell exception is present, and MSR_{EE}=1. Directed Privileged Doorbell exceptions are generated when Directed Privileged Doorbell messages (see Chapter 12) are received and accepted by the thread.

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 7.15 on page 1202.

Execution resumes at effective address 0x0000_0000_0000_0A00, possibly offset as specified in Figure 7.16.

7.5.14 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.

42:43 Set to indicate the LEV value specified by the *System Call* instruction that caused the interrupt, as follows.

LEV	$SRR1_{42:43}$
0	00
1	01
2	10
3*	undefined
* reserved LEV value	

44:47 Set to 0.

Others Loaded from the MSR.

MSR See Figure 7.15 on page 1202.

Execution resumes at effective address 0x0000_0000_0000_0C00, possibly offset as specified in Figure 7.16.

Programming Note

An attempt to execute an ***sc*** instruction with LEV=1 or LEV=2 in problem state, or an attempt to execute an ***sc*** instruction with LEV=2 in privileged non-hypervisor state, should be treated as a programming error.

An attempt to execute an ***sc*** instruction with LEV=2 when $SMFCTRL_E$=0 should be treated as a programming error.

7.5.15 Trace Interrupt

A Trace interrupt occurs when no higher priority exception exists and any instruction except ***rfid***, ***hrfid***, ***urfid***, ***rfscv,*** or a *Power-Saving Mode* instruction is successfully completed, provided any of the following is true:

- the instruction is ***mtmsr***[***d***] and 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. Thus a Trace interrupt never occurs for a *System Call* or *System Call Vectored* instruction, or for a *Trap* instruction that traps. The instruction that causes a Trace interrupt is called the “traced instruction”.

The following registers are set:

SRR0 Set to the effective address of the instruction that the thread would have attempted to execute next if no interrupt conditions were present.

SRR1

33 Set to 1.

34 Set to 0 if the traced instruction is a word instruction and to 1 if the traced instruction is a prefixed instruction.

35 Set to 1 if the the Trace interrupt is not the result of a CIABR match and the traced instruction is a *Load* instruction other than a *Load String* instruction with string length of 0 or is specified to be treated as a *Load* instruction; otherwise set to 0.

36 Set to 1 if the the Trace interrupt is not the result of a CIABR match and the traced instruction is a *Store* instruction other than a *Store String* instruction with string length of 0 or is specified to be treated as a *Store* instruction; otherwise set to 0.

42 Set to 0.

43 Set to 1 if the Trace interrupt is the result of a CIABR match.

44:47 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; for Trace interrupts caused by a CIABR match, not modified if MSR_{TE}=0b00; otherwise undefined.

SDAR For all Trace interrupts other than those caused by a CIABR match, set to the effective address of the storage operand (if any) of the traced instruction; for Trace interrupts caused by a CIABR match, not modified if MSR_{TE}=0b00; otherwise undefined.

If the state of the Performance Monitor is such that the Performance Monitor may be altering the SIAR and SDAR (i.e., if $MMCR0_{PMAE}$=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 7.15 on page 1202.

Execution resumes at effective address 0x0000_0000_0000_0D00, possibly offset as specified in Figure 7.16. For a Trace interrupt resulting from execution of an instruction that modifies the value of MSR_{IR}, MSR_{DR}, MSR_{HV}, $LPCR_{AIL}$, [] or $LPCR_{HAIL}$, the Trace interrupt vector location is based on the modified values.

Programming Note

The following instructions are not traced.

- ***rfid***
- ***hrfid***
- ***urfid***
- ***rfscv***
- ***sc***, ***scv***, and *Trap* instructions that trap
- *Power-Saving Mode* instructions
- other instructions that cause interrupts (other than Trace interrupts)
- the first instructions of any interrupt handler (applies to Branch and Single Step tracing; CIABR matches may still occur)
- instructions that are emulated by software

In general, interrupt handlers can achieve the effect of tracing these instructions.

Programming Note

If a ***wait*** instruction is executed when MSR_{TE}=0b10 or when the instruction causes a CIABR match, a Trace interrupt occurs immediately, with no suspension of instruction fetching or execution. (Architecturally, the suspension of instruction fetching and execution begins but is terminated immediately by the Trace interrupt.)

7.5.16 Hypervisor Data Storage Interrupt (HDSI)

A Hypervisor Data Storage interrupt occurs when no higher priority exception exists and a data access cannot be performed for any of the reasons listed below, under the conditions described immediately below.

In general, a Hypervisor Data Storage interrupt can occur only under the following conditions.

- HR=0, $MSR_{HV\ PR}\neq$0b10, and the value of the expression ($\neg MSR_{DR}$) | VPM [] is 1.
- HR=1 and partition-scoped translation, [] other than for the Process Table Entry when effLPID=0, prevents the data access from being performed.

[]

The exceptions to the preceding general statement are as follows.

- If HR=0 and $MSR_{HV\ PR}\neq$0b10, a security violation in accessing the PTEG during address translation causes a Hypervisor Data Storage interrupt regardless of the rest of the translation state.
- If $MSR_{HV\ PR}\neq$0b10 or MSR_{DR}=1, a translation mode mismatch causes a Hypervisor Data Storage interrupt regardless of the rest of the translation state.
- If HR=0, $MSR_{HV\ PR}\neq$0b10, MSR_{DR}=1, VPM=0, and $LPCR_{KBV}$=1, a Virtual Page Class Key Storage Protection exception causes a Hypervisor Data Storage interrupt, with HDSISR set according to the bit definitions for DSISR as set by a Data Storage interrupt, as described in more detail below.
- If HR=1, MSR_{DR}=1, effLPID=0, and either (a) an unsupported radix tree configuration is found in the Partition Table Entry, or (b) the effective address of the Process Table Entry cannot be translated to a host real address because no valid PTE is found that translates the effective address, a Hypervisor Data Storage interrupt occurs regardless of the rest of the translation state.
- If HR=1 and either $MSR_{HV\ PR}\neq$0b10 or MSR_{DR}=1, any of the following causes a Data Storage interrupt regardless of the rest of the translation state.
  - a Data Address Watchpoint match
  - an attempt to execute an AMO with a reserved function code
  - an attempt to copy from control memory, or to paste to control memory that is not properly configured

Architecture Note

If/when atomic R/C update is added for HPT, failures for these updates will always be directed to the hypervisor because the tables are managed by the hypervisor and there is nothing the OS can do. If the current style of storage interrupt description (with case screening at the beginning) still exists when this feature is added, it will be necessary to add text above to catch the case of VPM=0 since that typically results in a non-hypervisor storage interrupt.

Under the conditions described above, a Hypervisor Data Storage interrupt may occur for any of the following reasons.

- The virtual or guest real address of any byte of the storage location specified by a *Load* or *Store* instruction, or by an instruction that is treated as a *Load* or *Store* other than a *Cache Block Touch* instruction, cannot be translated to a host real address because no valid PTE was found that translates the virtual or guest real address.
- HR=0 and the virtual address of the Process Table Entry or Segment Table Entry Group cannot be translated because no valid PTE was found that translates the virtual address; or HR=1 and the effective address of the Process Table Entry (effLPID=0), or the guest real address of the Process Table Entry, a Page Directory Entry, or the Page Table Entry (effLPID≠0), cannot be translated because no valid PTE was found that translates the effective or guest real address.
- HR=1 and a Reference or Change bit update cannot be performed atomically in a partition-scoped PTE (including for the Process Table Entry, process-scoped PDE, or process-scoped PTE for a radix guest).
- []
- The effective address specified by a ***lq***, ***stq***, ***lwat***, ***ldat***, ***lbarx***, ***lharx***, ***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.

  Programming Note

  When nested Radix Tree translation is used to translate the effective address, these combinations of instruction and storage control attribute never cause a Hypervisor Data Storage interrupt (see Section 6.7.10.3 of Book III).

- Control memory is specified as the source of a ***copy*** instruction or an attempt is made to paste to control memory that is not properly configured. []
- The access violates Basic Storage Protection.
- The access violates Virtual Page Class Key Storage Protection.
- The access violates Radix Tree Translation Storage Protection.
- The access violates Secure Memory Protection.

  Programming Note

  A storage protection violation may be for the data access or for an associated implicit access to a table entry, other than the PATE, during address translation or when updating Reference and Change bits. ("Table entry" here includes STEGs, PTEGs, and PDEs. For data and table entries for which the host real address is not the result of address translation, the only possible storage protection violation is a Secure Memory Protection violation.) Note that when nested Radix Tree Translation is being performed, a violation of Radix Tree Translation Storage Protection causes an HDSI only when the process-scoped access authority permits the access but the partition-scoped access authority does not.

- Any of the following unsupported MMU configurations is found. (Note that these conditions may not be detected until the associated values are about to cause a functional problem for the processor.)
  - an unsupported radix tree configuration in the Partition Table Entry or in the partition-scoped tables, excluding the Process Table
  - a translation mode mismatch:
    - UPRT=0 when HR=1
    - LPID=0 and MSR_{HV}=0 when HR=1
    - HR=0 for LPID=0 when HR=1 for a non-zero LPID
- A Data Address Watchpoint match occurs. []
- An attempt is made to execute a *Load Atomic* or *Store Atomic* instruction with a reserved function code. []

Programming Note

A *Load Atomic* or *Store Atomic* instruction containing a reserved function code (see Figures 4.2 and 4.3 in Book II) is an invalid form of the instruction. Attempting to execute such a *Load Atomic* or *Store Atomic* instruction causes an HDSI (or DSI), rather than causing a Hypervisor Emulation Assistance interrupt or yielding boundedly undefined results, to permit the function code to be handled entirely by the memory agent, thereby facilitating the definition of additional function codes in the future if that proves desirable. Consistent with the usual handling of invalid instruction forms, this instance of [H]DSI occurs very early in the execution of the instruction, with the result that [hypervisor] data storage exceptions associated with computing and translating the EA do not occur and Reference and Change bits are not updated.

If a ***stbcx***., ***sthcx.***, ***stwcx.***, ***stdcx.***, or ***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 *Store* to the specified effective address would cause a Hypervisor Data Storage interrupt, it is implementation-dependent whether a Hypervisor Data Storage interrupt occurs.

If the XER specifies a length of zero for an indexed *Move Assist* instruction, a Hypervisor Data Storage interrupt does not occur.

The following registers are set:

HSRR0 Set to the effective address of the instruction that caused the interrupt.

HSRR1

- **33** Set to 0.
- **34** Set to 0 if the instruction that caused the interrupt is a word instruction and to 1 if the instruction that caused the interrupt is a prefixed instruction.
- **35:36** Set to 0.
- **42:47** Set to 0.
- **Others** Loaded from the MSR.

MSR See Figure 7.15.

HDSISR

- **32** Set to 0.
- **33** Set to 1 if the translation for an attempted access is not found in the Page Table; otherwise set to 0.
- **34:35** Set to 0.
- **36** Set to 1 if the access is not permitted by Figure 6.26 or the Read or Read/Write bits in Figure 6.27 as appropriate; otherwise set to 0.
- **37** Set to 1 if the access is due to a ***lq***, ***stq***, ***lwat***, ***ldat***, ***lbarx***, ***lharx***, ***lwarx***, ***ldarx***, ***lqarx***, ***stwat***, ***stdat***, ***stbcx.***, ***sthcx.***, ***stwcx.***, ***stdcx.***, or ***stqcx.*** instruction that addresses storage that is Write Through Required or Caching Inhibited; or if the access is due to a ***copy*** or ***paste.*** instruction that addresses storage that is caching inhibited; or if the access is due to a ***lwat***, ***ldat***, ***stwat***, or ***stdat*** instruction that addresses storage that is Guarded; otherwise set to 0.
- **38** Set to 1 for an explicit access for a *Store,* ***dcbz***, or *Load/Store Atomic* instruction; set to 1 when an implicit update of a process-scoped PTE fails due to lack of write authority or when an implicit update of the Change bit in the partition-scoped PTE fails; otherwise set to 0.
- **39:40** Set to 0.
- **41** Set to 1 if a Data Address Watchpoint match occurs; otherwise set to 0.
- **42** Set to 1 if the access is not permitted by Virtual Page Class Key Protection; otherwise set to 0.
- **43** Set to 1 if the access is not permitted by Secure Memory Protection; otherwise set to 0.
- **44** Set to 1 if an unsupported MMU configuration is found during the translation process; otherwise set to 0.
- **45** Set to 1 if an attempt to atomically set a Reference or Change bit fails; otherwise set to 0.

  Programming Note

  The number of attempts hardware makes to atomically set Reference and Change bits before triggering this exception is implementation dependent. For example, the POWER9 processor makes no attempt. Software may still support the atomic update programming model to get performance benefits such as those described in Section 6.7.12.

- **46** Set to 1 if [] the guest real address of a page directory entry, page table entry, or process table entry cannot be translated; or [] the virtual address of a Process Table Entry or Segment Table Entry Group cannot be translated; otherwise set to 0.
- **47:59** Set to 0.
- **60** Set to 1 if control memory is specified as the source of a ***copy*** instruction [] or an attempt is made to paste to control memory that is not properly configured; [] 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 control memory is not properly configured.

61 Set to 1 if an attempt is made to execute a *Load Atomic* or *Store Atomic* instruction specifying a reserved function code; otherwise set to 0.

62:63 Set to 0.

HDAR Set to the effective address or portion of the VPN of a storage element, or undefined, as described in the following list. The list should be read from the top down; the HDAR is set as described by the first item that corresponds to an exception that is reported in the HDSISR. For example, if a *Load Word* instruction causes a storage protection violation and a Data Address Watchpoint match (and both are reported in the HDSISR), the HDAR is set to the effective address of a byte in the first aligned doubleword for which access was attempted in the page that caused the exception.

- undefined, for *Load Atomic* or *Store Atomic* instruction specifying a reserved function code
- undefined, when $HDSISR_{60}$=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. []
- EA, when a Hypervisor Data Storage exception occurs for reasons other than a Data Address Watchpoint match, of
  - a byte in the block that caused the exception, for a *Cache Management* instruction
  - a byte in the first aligned doubleword for which access was attempted in the page that caused the exception, for a *Load* or *Store* instruction or an instruction that is treated as a *Load* or *Store* ("first" refers to address order: see Section 7.7).
- EA of the first byte of overlap between the operand and the matching watched range, for a Data Address Watchpoint match

For the cases in which the HDAR is specified above to be set to an effective address, if the interrupt occurs in 32-bit mode the high-order 32 bits of the HDAR are set to 0.

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, K_s, K_p, N, C, L, and LP values from the segment descriptor that translated the access or indicated the base of the table, or undefined, as described in the following list. When HR=1 and 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 a reserved function code
- undefined, when $HDSISR_{60}$=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 (group) when a Process Table Entry or Segment Table Entry Group virtual address cannot be translated (the rest of the segment descriptor is implied, or, for the base page size, comes from the Partition Table Entry or Process Table Entry, respectively).
- the guest real address of the process-scoped PDE or PTE or Process Table Entry when a Reference or Change bit in the partition-scoped PTE mapping the process-scoped PDE or PTE or Process Table Entry cannot be set atomically
- the guest real address of the storage element when a Reference or Change bit in the partition-scoped PTE cannot be set atomically
- the guest real address of the storage element, Process Table Entry, Page Directory Entry, or Page Table Entry (depending on which partition-scoped table has the flaw) for an unsupported radix tree configuration for the partition-scoped tables (the effective address for other cases of the unsupported 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, 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
- 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 a Hypervisor Data Storage exception occurs due to an attempt to execute a *Load Atomic* or *Store Atomic* instruction with a reserved function code, the only exception bit that is set in the HDSISR is bit 61 and the remainder of this paragraph does not apply. Otherwise, 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 Virtual Page Class Key Storage Protection exception that occurs when the thread is not in hypervisor state ($MSR_{HV\ PR} \neq 0b10$), $LPCR_{KBV}$=1, and Virtualized Partition Memory is disabled by VPM=0, the other exceptions are actually Data Storage exceptions, and HDSISR is set according to the bit definitions for DSISR as set by a Data Storage interrupt.

Programming Note

A Virtual Page Class Key Storage Protection exception that occurs when the thread is not in hypervisor state ($MSR_{HV\ PR} \neq 0b10$), $LPCR_{KBV}$=1, and Virtualized Partition Memory is 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 7.16.

Architecture Note

In general, if an instruction that accesses data storage causes an interrupt, and the attempted access should be emulated by the interrupt handler, the interrupt should be an Alignment interrupt rather than a [Hypervisor] Data Storage interrupt.

Engineering Note

For initial hardware debug it is often useful to run with cache disabled. In some ways cache disabled mode is similar to Caching Inhibited storage. Although ***lq*** and ***stq*** need not be supported by an implementation for storage that is Caching Inhibited, support for ***lq*** and ***stq*** with cache disabled should be considered.

7.5.17 Hypervisor Instruction Storage Interrupt (HISI)

A Hypervisor Instruction Storage interrupt occurs when no higher priority exception exists and the next instruction to be executed cannot be fetched for any of the reasons listed below, under the conditions described immediately below.

In general, a Hypervisor Instruction Storage interrupt can occur only under the following conditions.

- HR=0, $MSR_{HV\ PR} \neq 0b10$, and the value of the expression $(\neg MSR_{IR})$ | VPM [] is 1. []
- HR=1 and partition-scoped translation, other than for the Process Table Entry when effLPID=0, prevents the next instruction to be executed from being fetched.

The exceptions to the preceding general statement are as follows.

- If HR=0 and $MSR_{HV\ PR} \neq 0b10$, a security violation in accessing the PTEG during address translation causes a Hypervisor Instruction Storage interrupt regardless of the rest of the translation state.
- If $MSR_{HV\ PR} \neq 0b10$ or MSR_{IR}=1, a translation mode mismatch causes a Hypervisor Instruction Storage interrupt regardless of the rest of the translation state.
- If HR=1, MSR_{IR}=1, effLPID=0, and either (a) an unsupported radix tree configuration is found in the Partition Table Entry, or (b) the effective address of the Process Table Entry cannot be translated to a host real address because no valid PTE is found that translates the effective address, a Hypervisor Instruction Storage interrupt occurs regardless of the rest of the translation state.
- If the instruction is a prefixed instruction and is in storage that is Caching Inhibited, attempting to fetch it causes an Instruction Storage interrupt regardless of the rest of the translation state.

Architecture Note

If/when atomic R/C update is added for HPT, failures for these updates will always be directed to the hypervisor because the tables are managed by the hypervisor and there is nothing the OS can do. If the current style of storage interrupt description (with case screening at the beginning) still exists when this feature is added, it will be necessary to add text above to catch the case of VPM=0 since that typically results in a non-hypervisor storage interrupt.

Under the conditions described above, a Hypervisor Instruction Storage interrupt may occur for any of the following reasons.

- The virtual or guest real [] address of the next instruction to be executed cannot be translated to a host real address because no valid PTE was found that translates the virtual or guest real address.
- HR=0 and the virtual address of the Process Table Entry or Segment Table Entry Group cannot be translated because no valid PTE was found that translates the virtual address; or HR=1 and the effective address of the Process Table Entry (effLPID=0), or the guest real address of the Process Table Entry, a Page Directory Entry, or the Page

Table Entry (effLPID≠0), cannot be translated because no valid PTE was found that translates the effective or guest real address.

- HR=1 and a Reference or Change bit update cannot be performed atomically in a partition-scoped PTE (including for the Process Table Entry, process-scoped PDE, or process-scoped PTE for a radix guest).

Programming Note

When failure to set a Reference or Change bit is reported, whether the Change bit must be set is indicated by whether $HSRR1_{47}$ is set to 1. Whether an access authority failure was caused by absence of write authority is determined similarly.

- The access violates Basic Storage Protection.
- The access violates Virtual Page Class Key Storage Protection.
- The access violates Radix Tree Translation Storage Protection.
- The access violates Secure Memory Protection.

Programming Note

A storage protection violation may be for the instruction fetch or for an associated implicit access to a table entry, other than the PATE, during address translation or when updating Reference and Change bits. ("Table entry" here includes STEGs, PTEGs, and PDEs. For instructions and table entries for which the host real address is not the result of address translation, the only possible storage protection violation is a Secure Memory Protection violation.) Note that when nested Radix Tree Translation is being performed, a violation of Radix Tree Translation Storage Protection causes an HISI only when the process-scoped access authority permits the access but the partition-scoped access authority does not.

- Any of the following unsupported MMU configurations is found. (Note that these conditions may not be detected until the associated values are about to cause a functional problem for the processor.)
  - an unsupported radix tree configuration in the Partition Table Entry or in the partition-scoped tables, excluding the Process Table
  - a translation mode mismatch:
    - UPRT=0 when HR=1
    - LPID=0 and MSR_{HV}=0 when HR=1
    - HR=0 for LPID=0 when HR=1 for a non-zero LPID

The following registers are set:

HSRR0 Set to the effective address of the instruction that the thread would have attempted to execute next if no interrupt conditions were present (if the interrupt occurs on attempting to fetch a branch target, HSRR0 is set to the branch target address).

HSRR1

33 Set to 1 if the translation for an attempted access is not found in the Page Table; otherwise set to 0.

34 Set to 0.

35 Set to 1 if the access is to No-execute (as indicated by the N bit in the Segment Table Entry and HPT PTE or the Execute bit in the EAA field of the Radix PTE) or Guarded storage; otherwise set to 0.

36 Set to 1 if the access is not permitted by Figure 6.26 or the Execute bit in Figure 6.27 as appropriate; otherwise set to 0.

42 Set to 1 if the access is not permitted by Virtual Page Class Key Protection; otherwise set to 0.

43 Set to 1 if the access is not permitted by Secure Memory Protection; otherwise set to 0.

44 Set to 1 if an unsupported MMU configuration is found during the translation process; otherwise set to 0.

45 Set to 1 if an attempt to atomically set a Reference or Change bit fails; otherwise set to 0.

Programming Note

The number of attempts hardware makes to atomically set Reference and Change bits before triggering this exception is implementation dependent. For example, the POWER9 processor makes no attempt. Software may still support the atomic update programming model to get performance benefits such as those described in Section 6.7.12.

46 Set to 1 if [] the guest real address real address of a Page Directory Entry, Page Table Entry, or Process Table Entry cannot be translated; or [] the virtual address of a Process Table Entry or Segment Table Entry Group cannot be translated; otherwise set to 0.

47 Set to 1 if the operation that caused the exception is 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 is 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 [] a Process Table Entry or Segment Table Entry Group virtual address cannot be translated. [] May be set spuriously in other cases.

ASDR When HR=0, loaded with VSID, B, K_s, K_p, N, C, L, and LP values from the segment descriptor that translated the access or indicated the base of the table, or undefined, as described in the following list. When HR=1 and nested translation is taking place, loaded with the guest real address down to bit 51 of the instruction or table entry, or undefined, as described in the following list. The 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 HSRR1.

- the guest real address of the table entry when a Process Table or process-scoped Page Directory or Page Table Entry guest real address cannot be translated or the VSID of the table entry (group) when a Process Table Entry or Segment Table Entry Group virtual address cannot be translated (the rest of the segment descriptor is implied, or, for the base page size, comes from the Partition Table Entry or Process Table Entry, respectively).
- the guest real address of the process-scoped PDE or PTE or Process Table Entry when a Reference or Change bit in the partition-scoped PTE mapping the process-scoped PDE or PTE or Process Table Entry cannot be set atomically
- the guest real address of the instruction when a Reference or Change bit in the partition-scoped PTE cannot be set atomically
- the guest real address of the instruction, Process Table Entry, Page Directory Entry, or Page Table Entry (depending on which partition-scoped table has the flaw) for an unsupported radix tree configuration for the partition-scoped tables (the effective address for other cases of the unsupported MMU configuration exception will be found in HSRR0)
- the guest real address of the process-scoped PTE when an attempt is made to set a Reference bit without 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 7.15.

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_0E20, possibly offset as specified in Figure 7.16.

7.5.18 Hypervisor Emulation Assistance Interrupt

A Hypervisor Emulation Assistance interrupt is generated when execution is attempted of an illegal instruction, or of a reserved instruction or an instruction that is not provided by the implementation. It is also generated under the following conditions.

- When $MSR_{HV\ PR}$=0b00 and $LPCR_{EVIRT}$=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 or that specifies PTCR, DAWRn, DAWRXn, or CIABR when those SPRs are ultravisor privileged for the operation.
- When $MSR_{S\ HV\ PR}$ = 0b010, execution is attempted of an ***mtspr*** or ***mfspr*** instruction that specifies PTCR, DAWRn, DAWRXn, or CIABR when those SPRs are ultravisor privileged for the operation.
- When MSR_{PR}=1, execution is attempted of an ***mtspr*** or ***mfspr*** instruction that specifies an SPR with spr_0=0 that is not provided by the implementation, or of an ***mtspr*** instruction that specifies a Performance Monitor register in group A when $MMCR0_{PMCC}$=0b00.
- When MSR_{PR}=0, execution is attempted of an ***mtspr*** or ***mfspr*** instruction that specifies SPR 0, 4, 5, or 6.
- When MSR_{PR}=0 and $LPCR_{EVIRT}$=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.

Engineering Note

The requirement that an attempt to execute a reserved instruction, an ***mtspr*** instruction specifying SPR 0, or an ***mfspr*** instruction specifying SPR 0, 4, 5, or 6 causes a Hypervisor Emulation Assistance interrupt results from the need to support legacy software that uses POWER instructions and accesses POWER SPRs that are not in Power ISA.

The following registers are set:

HSRR0 Set to the effective address of the instruction that caused the interrupt.

HSRR1

33 Set to 0.

34 Set to 0 if the instruction that caused the interrupt is a word instruction (or a prefixed instruction when prefixed instructions are unavailable based on the PCR setting) and to 1 if the instruction that caused the interrupt is a prefixed instruction.

35:36 Set to 0.

42:44 Set to 0.

45 Set to 1 for an attempt, when $MSR_{HV\ PR}$ = 0b00 and $LPCR_{EVIRT}$=1, to execute a hypervisor privileged instruction or an ***mtspr*** or ***mfspr*** instruction that specifies an SPR that is hypervisor privileged for the operation or that specifies PTCR, DAWRn, DAWRXn, or CIABR when they are ultravisor privileged for the operation, or for an attempt when $MSR_{S\ HV\ PR}$ = 0b010 to execute an ***mtspr*** or ***mfspr*** instruction that specifies PTCR, DAWRn, DAWRXn, or CIABR when they are ultravisor privileged for the operation; otherwise set to 0.

46:47 Set to 0.

Others Loaded from the MSR.

MSR See Figure 7.15 on page 1202.

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 7.16.

Programming Note

This Programming Note illustrates how Hypervisor Emulation Assistance interrupts should be handled by software, including in environments that support nested hypervisors. For simplicity, this Programming Note ignores effects of the SMF facility (equivalently, assumes that $SMFCTRL_E$=0).

In this Note, "the hypervisor" may be the hypervisor to which hardware passes control when a Hypervisor Emulation Assistance interrupt occurs or, in an environment that supports nested hypervisors, may be a nested hypervisor. The hypervisor to which hardware passes control when a Hypervisor Emulation Assistance interrupt occurs is here called the "level 0 hypervisor," and is the only level of hypervisor that runs with $MSR_{HV\ PR}$=0b10 and that can access hypervisor resources directly; nested hypervisors run with $MSR_{HV\ PR}$=0b00 and their attempts to access hypervisor resources are virtualized by a higher-level hypervisor as described below. In this Note, the hypervisor receiving the Hypervisor Emulation Assistance interrupt (which may have been passed from a higher-level hypervisor as described below) is called the "level N hypervisor." This Note assumes that $LPCR_{EVIRT}$=1 if nested hypervisors are used. (A Hypervisor Emulation Assistance interrupt can set $HSRR1_{45}$ to 1 only when $LPCR_{EVIRT}$=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.
  - $HSRR1_{45}$=0: Emulate the instruction, recover from the error, or terminate this hypervisor, as appropriate.
  - $HSRR1_{45}$=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.
    - $HSRR1_{45}$=0: Pass control to the level N+1 hypervisor as if the instruction had caused a Hypervisor Emulation Assistance interrupt (with $HSRR1_{45}$=0) to that hypervisor.
    - $HSRR1_{45}$=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 $HSRR1_{45}$=1) to that hypervisor.
  - The program most recently dispatched by the level N hypervisor is an operating system.
    - $HSRR1_{45}$=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 Section 7.5.9.
    - $HSRR1_{45}$=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 Section 7.5.9.
  - The program most recently dispatched by the level N hypervisor is an application program.
    - $HSRR1_{45}$=0: Emulate the instruction if appropriate; otherwise terminate the application program.
    - $HSRR1_{45}$=1: Cannot occur.

The preceding description implicitly assumes that any nested hypervisors being tested will, when run at level 0, be run on processors that support the same version of the architecture as the processor on which they are being tested. If instead they will be run on processors that support a newer version of the architecture, the level 0 hypervisor should behave as described above if the interrupt is caused by an instruction that is unchanged between the two architecture versions. However, if the interrupt is caused by an instruction that differs between the two architecture versions (e.g., an instruction that is added by the newer version of the architecture), the level 0 hypervisor should emulate the behavior of the newer processor, rather than, for example, passing the interrupt to a level 1 hypervisor.

Other uses of nested hypervisors are also possible. For example, software that is designed to interact, nearly simultaneously, with the hypervisor instance that is running on each of many processors could be tested on a single processor by running multiple level 1 hypervisors under a single level 0 hypervisor.

[]

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 7.5.9, "Program Interrupt" on page 1215.

Similarly, if a Hypervisor Emulation Assistance interrupt occurs with $HSRR1_{45}$=1 when the thread is in privileged non-hypervisor state, for an instruction that the hypervisor does not virtualize, the hypervisor should pass control to the operating system as if the instruction had caused a Privileged Instruction type Program interrupt, as described in another Programming Note near the end of Section 7.5.9, "Program Interrupt" on page 1215.

Programming Note

In versions of the architecture that precede V. 3.1B, an attempt when MSR_{PR}=0 to execute an ***mtspr*** or ***mfspr*** instruction specifying an SPR that was not implemented (with the exception of SPR 0 for ***mtspr*** 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 ***mtspr*** 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

Engineering Note

The hypervisor should not be expected to emulate instructions that have storage operands. (For environments that use HPT translation, it would be difficult for the hypervisor to ensure that the address translation for the storage operands is the same as would have been used for the interrupted program. Related difficulties exist for level N>0 hypervisors in environments that use Radix on Radix translation. Complexity of design and verification is avoided by never expecting the hypervisor to emulate instructions that have storage operands, even though such emulation would not be difficult in environments that use Radix on Radix translation and do not use nested hypervisors.)

7.5.19 Hypervisor Maintenance Interrupt

A Hypervisor Maintenance interrupt occurs when no higher priority exception exists, a Hypervisor Maintenance exception exists (a bit in the HMER is set to one), the exception is enabled in the HMEER, and the value of the following expression is 1.

$(MSR_{EE} \mid \neg(MSR_{HV}) \mid MSR_{PR})$

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 7.15 on page 1202.

HMER See Section 7.2.10 on page 1194.

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

Because the value of MSR_{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.

[]

Engineering Note

Because HMIs do not occur when the thread is in hypervisor state with MSR_{EE}=0, events which critically impact hypervisor state or which are exposed to further propagation within or external to the system must use a stronger notification mechanism such as Machine Check. Examples include scenarios where forward progress to even attempt to poll would be impossible.

Engineering Note

HMIs may be presented to threads other than those affected by the exceptions in order to simplify the distribution mechanism.

7.5.20 Directed Hypervisor Doorbell Interrupt

A Directed Hypervisor Doorbell interrupt occurs when no higher priority exception exists, a Directed Hypervisor Doorbell exception is present, and the value of the following expression is 1.

$$(MSR_{EE} \mid \neg(MSR_{HV}) \mid MSR_{PR})$$

Directed Hypervisor Doorbell exceptions are generated when Directed Hypervisor Doorbell messages (see Chapter 12) 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 7.15 on page 1202.

Execution resumes at effective address 0x0000_0000_0000_0E80, possibly offset as specified in Figure 7.16.

Programming Note

Because the value of MSR_{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.

7.5.21 Hypervisor Virtualization Interrupt

A Hypervisor Virtualization interrupt occurs when no higher priority exception exists, a Hypervisor Virtualization exception exists, and the value of the following equation is 1.

$$(MSR_{EE} \mid \neg(MSR_{HV}) \mid MSR_{PR}) \ \& \ HVICE$$

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 7.15 on page 1202.

Execution resumes at effective address 0x0000_0000_0000_0EA0, possibly offset as specified in Figure 7.16.

7.5.22 Performance Monitor Interrupt

A Performance Monitor interrupt occurs when no higher priority exception exists, a Performance Monitor exception exists, event-based branches are disabled ($MMCR0_{EBE}$=0), and MSR_{EE}=1, and either $HFSCR_{PM}$=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 Set to 0.

42:47 Set to 0.

Others Loaded from the MSR.

MSR See Figure 7.15 on page 1202.

Execution resumes at effective address 0x0000_0000_0000_0F00, possibly offset as specified in Figure 7.16.

7.5.23 Vector Unavailable Interrupt

A Vector Unavailable interrupt occurs when no higher priority exception exists, an attempt is made to execute a Vector instruction (including Vector loads, stores, and moves), and MSR_{VEC}=0.

The following registers are set:

SRR0 Set to the effective address of the instruction that caused the interrupt.

SRR1

33 Set to 0.

34 Set to 0 if the instruction that caused the interrupt is a word instruction and to 1 if the instruction that caused the interrupt is a prefixed instruction.

35:36 Set to 0.

42:47 Set to 0.

Others Loaded from the MSR.

MSR See Figure 7.15 on page 1202.

Execution resumes at effective address 0x0000_0000_0000_0F20, possibly offset as specified in Figure 7.16.

7.5.24 VSX Unavailable Interrupt

A VSX Unavailable interrupt occurs when no higher priority exception exists, an attempt is made to execute a VSX instruction (including VSX loads, stores, and moves), and MSR_{VSX}=0.

The following registers are set:

SRR0 Set to the effective address of the instruction that caused the interrupt.

SRR1

33 Set to 0.

34 Set to 0 if the instruction that caused the interrupt is a word instruction and to 1 if the instruction that caused the interrupt is a prefixed instruction.

35:36 Set to 0.

42:47 Set to 0.

Others Loaded from the MSR.

MSR See Figure 7.15 on page 1202.

Execution resumes at effective address 0x0000_0000_0000_0F40, possibly offset as specified in Figure 7.16.

7.5.25 Facility Unavailable Interrupt

A Facility Unavailable interrupt occurs when no higher priority exception exists, and one of the following occurs.

- a facility is accessed in problem state when it has been made unavailable by the FSCR
- a Performance Monitor register is accessed or a ***clrbhrb*** or ***mfbhrbe*** instruction is executed in problem state when it has been made unavailable by MMCR0.

The following registers are set:

SRR0 Set to the effective address of the instruction that caused the interrupt.

SRR1

33 Set to 0.

34 Set to 0 if the instruction that caused the interrupt is a word instruction and to 1 if the instruction that caused the interrupt is a prefixed instruction.

35:36 Set to 0.

42:47 Set to 0.

Others Loaded from the MSR.

MSR See Figure 7.15 on page 1202.

FSCR

0:7 See Section 7.2.12 on page 1194.

Others Not changed.

Execution resumes at effective address 0x0000_0000_0000_0F60, possibly offset as specified in Figure 7.16.

7.5.26 Hypervisor Facility Unavailable Interrupt

A Hypervisor Facility Unavailable interrupt occurs when no higher priority exception exists, and one of the following occurs.

- a facility is accessed in problem or privileged non-hypervisor states when it has been made unavailable by the HFSCR.
- The ***stop*** instruction is executed in privileged non hypervisor state when any of the following conditions exist.

  $PSSCR_{EC}$=1
  $PSSCR_{ESL}$=1
  $PSSCR_{MTL} > PSSCR_{PSLL}$
  $PSSCR_{RL} > PSSCR_{PSLL}$

The following registers are set:

HSRR0 Set to the effective address of the instruction that caused the interrupt.

HSRR1

33 Set to 0.

34 Set to 0 if the instruction that caused the interrupt is a word instruction and to 1 if the instruction that caused the interrupt is a prefixed instruction.

35:36 Set to 0.

42:47 Set to 0.

Others Loaded from the MSR.

MSR See Figure 7.15 on page 1202.

HFSCR

0:7 See Section 7.2.13 on page 1195.

Others Not changed.

Execution resumes at effective address 0x0000_0000_0000_0F80, possibly offset as specified in Figure 7.16.

Programming Note

The Hypervisor Facility Unavailable interrupt handler should either (a) make the facility, the attempted use of which caused the interrupt, available, or (b) pass control to the operating system as if the instruction that caused the interrupt had instead caused an "Illegal Instruction type Program interrupt", as described in a Programming Note near the end of Section 7.5.9. Specifically, for choice (b) the hypervisor should pass control to the operating system at the operating system's Program interrupt vector location, with all registers (SRR0, SRR1, MSR, GPRs, etc.) set as if the instruction had caused a Privileged Instruction type Program interrupt, except with $SRR1_{44:45}$ set to 0b10. (This behavior is the same as that provided by the Hypervisor Emulation Assistance interrupt handler when that interrupt is caused by an illegal instruction or by ***mt/fspr*** specifying an undefined SPR number.) In general this behavior by the Hypervisor Facility Unavailable interrupt handler provides to the operating system the appearance that the instructions in the facility are illegal instructions and that the SPRs in the facility correspond to undefined SPR numbers. The cases in which it does not provide this appearance are as follows.

1. **privileged instruction executed in problem state**

   Because Privileged Instruction type Program interrupt has higher priority than Hypervisor Facility Unavailable interrupt, an attempt in problem state to execute a privileged instruction made unavailable by the HFSCR will cause a Privileged Instruction type Program interrupt to the operating system, rather than a Hypervisor Facility Unavailable interrupt, so the hypervisor will not have opportunity to make the instruction appear to be illegal. (It may be useful to note that the handling described in choice (b) above together with the behavior of this case provides behavior, in problem state, that is equivalent to the behavior that would be obtained by making the facility unavailable by means of the PCR.)

2. ***mtspr*/*mfspr* executed in privileged non-hypervisor state when $LPCR_{EVIRT}$=0**

   mtspr*/*mfspr specifying an undefined SPR number (other than 0, 4, 5, 6) and executed in privileged non-hypervisor state when $LPCR_{EVIRT}$=0 will be treated as a no-op. If instead the SPR number is defined and the SPR is made unavailable by the HFSCR a Hypervisor Facility Unavailable interrupt will occur, and there is no easy way for the interrupt handler to determine that the interrupting instruction is ***mt/fspr*** and hence should be treated as a no-op. (Hypervisor Facility Unavailable interrupt does not set HEIR.) Passing control to the operating system's Program interrupt handler in the manner described above is preferable to incurring the software complexity and performance cost of emulating the no-op behavior.

7.5.27 System Call Vectored Interrupt

A System Call Vectored interrupt occurs when a *System Call Vectored* instruction is executed.

The following registers are set:

LR Set to the effective address of the instruction following the System Call Vectored instruction.

CTR

33:36 undefined

42:47 undefined

Others Loaded from corresponding bits of the MSR.

MSR See Figure 7.15 on page 1202.

Execution resumes at the effective address specified in Figure 7.16.

Programming Note

When the System Call Vectored interrupt results in MSR_{IR} being 1 or MSR_{HV} 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 MSR_{RI} at the end of Section 7.4.3 does not apply to the System Call Vectored interrupt handler (the System Call Vectored interrupt does not alter MSR_{RI}).

7.5.28 Directed Ultravisor Doorbell Interrupt

A Directed Ultravisor Doorbell interrupt occurs when no higher priority exception exists, $SMFCTRL_E$=1, a Directed Ultravisor Doorbell exception is present, and the value of the following expression is 1.

$(MSR_{EE} \mid \neg(MSR_{S\ HV\ PR}=0b110))$

Directed Ultravisor Doorbell exceptions are generated when Directed Ultravisor Doorbell messages (see Chapter 12) are received and accepted by the thread.

The following registers are set:

USRR0 Set to the effective address of the instruction that the thread would have attempted to execute next if no interrupt conditions were present.

USRR1

33:36 Set to 0.

42:47 Set to 0.

Others Loaded from the MSR.

MSR See Figure 7.15 on page 1202.

Execution resumes at effective address 0x0000_0000_0000_0FA0.

7.6 Partially Executed Instructions

If a Data Storage, Data Segment, Alignment, system-caused, or imprecise exception occurs while a *Load* or *Store* instruction is executing, the instruction may be aborted. In such cases the instruction is not completed, but may have been partially executed in the following respects.

- Some of the bytes of the storage operand may have been accessed, except that if access to a given byte of the storage operand would violate storage protection, that byte is neither copied to a register by a *Load* instruction nor modified by a *Store* instruction. Also, the rules for storage accesses given in Section 6.8.1, “Guarded Storage” and in Section 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 6.7.12.
- 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.

7.7 Exception Ordering

Since multiple exceptions can exist at the same time and the architecture does not provide for reporting more than one interrupt at a time, the generation of more than one interrupt is prohibited. Some exceptions, such as the Mediated External exception, persist and can be deferred. However, other exceptions would be lost if they were not recognized and handled when they occur. For example, 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 occur as if the storage operand were accessed one byte at a time in order of increasing effective address (with the obvious caveat if the operand includes both the maximum effective address and effective address 0). (The required ordering of exceptions on components of non-atomic accesses does not extend to the performing of the component accesses in the event of an exception. For example, if byte n causes a data storage exception, it is not necessarily true that the access to byte n-1 has been performed.)

7.7.1 Unordered Exceptions

With one exception, the exceptions listed here are unordered, meaning that they may occur at any time regardless of the state of the interrupt processing mechanism. These exceptions are recognized and processed when presented. The exception is that a Machine Check caused by an attempt to access control memory as other than an operand of ***copy*** or ***paste.*** is ordered similarly to the corresponding type of storage access exception. (Note that this results in two different orderings. The one for instruction fetch occurs early, as item 2 in the "Instruction-Caused and Precise" list. The one for data access appears later, within case 3 of the appropriate "Function-Dependent" listings.)

1. System Reset
2. Machine Check except for those caused by an invalid attempt to access control memory

7.7.2 Ordered Exceptions

The exceptions listed here are ordered with respect to the state of the interrupt processing mechanism. With one exception, in the following list the hypervisor forms of the Data Storage and Instruction Storage exceptions can be substituted for the non-hypervisor forms since the hypervisor forms cannot be caused by the same instruction and have the same ordering. The exception is that Virtual Page Class Key Storage Protection exceptions that occur when $LPCR_{KBV}$=1 and Virtualized Partition Memory is disabled by VPM=0 cause only a Hypervisor Data Storage exception (and never a Data Storage exception).

In the list below for Instruction-Caused and Precise exceptions, for prefixed instructions items 1, 2, and 6 apply to the first word of the instruction, and determination of whether the exceptions of items 3-5 and 7 occur is based on the first word of the instruction. (Items 3-7 can occur only for prefixed instructions.)

Programming Note

For a prefixed instruction there is no need to determine whether the exceptions of items 1, 2, and 6 can be caused by the second word of the instruction.

- If the instruction is correctly aligned (i.e., is at an effective address that is not equal to 60 modulo 64), the determination would be the same for the second word as for the first word.
- If the instruction is not correctly aligned, an exception will occur due to one of items 1–7 applied to the first word, and the corresponding interrupt obviates the need for hardware to access the second word.

System-Caused or Imprecise

1. Program
   – Imprecise Mode Floating-Point Enabled Exception
2. Directed Ultravisor Doorbell
3. Hypervisor Maintenance
4. Hypervisor Virtualization, External, [Hypervisor] Decrementer, Performance Monitor, Directed Privileged Doorbell, Directed Hypervisor Doorbell

Instruction-Caused and Precise

1. Instruction Segment
2. [Hypervisor] Instruction Storage other than case described in item 6
3. Hypervisor Emulation Assistance due to PCR making all prefixed instructions unavailable
4. Hypervisor Facility Unavailable due to HFSCR making all prefixed instructions unavailable
5. Facility Unavailable due to HFSCR making all prefixed instructions unavailable
6. [Hypervisor] Instruction Storage for prefixed instruction in Caching Inhibited storage
7. Alignment for incorrectly aligned prefixed instruction
8. Machine Check for invalid control memory access
9. Other Hypervisor Emulation Assistance or Program (Privileged Instruction)
10. Function-Dependent

10.a Fixed-Point and Branch
- 1 Hypervisor Facility Unavailable
- 2 Facility Unavailable
- 3a Program – Trap
- 3b System Call or System Call Vectored
- 3c.1 Data Storage for the case of *Fixed-Point Load or Store Caching Inhibited* instructions with MSR_{DR}=1 or the case of a reserved function code for an Atomic Memory Operation
- 3c.2 all other Data Storage, Hypervisor Data Storage, [Hypervisor] Data Segment, Machine Check for invalid control memory access, or Alignment
- 4 Trace

10.b Floating-Point
- 1 Hypervisor Facility Unavailable
- 2 Floating Point Unavailable
- 3a Program - Precise Mode Floating-Point Enabled Exception
- 3b [Hypervisor] Data Storage, [Hypervisor] Data Segment, Machine Check for invalid control memory access, or Alignment
- 4 Trace

10.c Vector
- 1 Hypervisor Facility Unavailable
- 2 Vector Unavailable
- 3a [Hypervisor] Data Storage, [Hypervisor] Data Segment, Machine Check for invalid control memory access, or Alignment
- 4 Trace

10.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 control memory access, or Alignment
- 4 Trace

10.e Other Instructions
- 1 Hypervisor Facility Unavailable
- 2 Facility Unavailable
- 3a [Hypervisor] Data Storage, [Hypervisor] Data Segment, Machine Check for invalid control memory 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

7.8 Event-Based Branch Exception Ordering

Event-based exceptions are not ordered because they can occur simultaneously. Whenever an event-based exception occurs and the exception is enabled, the corresponding "exception occurred" bit in the BESCR is set to 1. See Section 6.2.1 of Book II.

7.9 Interrupt Priorities

This section describes the relationship of nonmaskable, maskable, precise, and imprecise interrupts. In the following descriptions, the interrupt mechanism waiting for all possible exceptions to be reported includes only exceptions caused by previously initiated instructions (e.g., it does not include waiting for the Decrementer to step through zero). The exceptions are listed in order of highest to lowest priority. The phrase "corresponding interrupt" means the interrupt having the same name as the exception unless the thread is in power-saving mode, in which case the phrase means the System Reset interrupt.

Unless otherwise stated or obvious from context, it is assumed below that one of the following conditions is satisfied.

- The thread is not in power-saving mode and the interrupt, unless it is the Machine Check interrupt, is not disabled. (For the Machine Check interrupt no assumption is made regarding enablement.)
- The thread is in power-saving mode and the exception is enabled to cause exit from the mode.

With one exception, in the following list the hypervisor forms of the Data Storage and Instruction Storage exceptions can be substituted for the non-hypervisor forms since the hypervisor forms cannot be caused by the same instruction and have the same priority. The exception is that exceptions caused by Virtual Page Class Key Storage Protection exceptions that occur when $LPCR_{KBV}$=1 and Virtualized Partition Memory is disabled by VPM=0 cause only a Hypervisor Data Storage exception (and never a Data Storage exception).

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 control memory as other than an operand of ***copy*** or ***paste.*** is prioritized similarly to the corresponding type of storage access exception. (Note that this results in two different priorities. The one for data access is higher, appearing as item e or f in the listings for different types of *Load* and *Store* instructions. The one for instruction fetch is lower, in category K.)

   Once the Machine Check interrupt is generated, no nonmaskable interrupts are generated due to exceptions caused by instructions issued prior to the generation of this interrupt.

3. Instruction-Caused and Precise

   This exception is the third highest priority exception. When this exception is created, the interrupt mechanism waits for all possible Imprecise exceptions to be reported. It then generates the appropriate ordered interrupt if no higher priority exception exists when the interrupt is to be generated. Within this category a particular instruction may present more than a single exception. When this occurs, those exceptions are ordered in priority as indicated in the following lists. Where [Hypervisor] Data Storage, Data Segment, and Alignment exceptions are listed in the same item they have equal priority (i.e., the hardware may generate any one of the three interrupts for which an exception exists).

   A. Fixed-Point Loads and Stores
      a. These exceptions are mutually exclusive and have the same priority:
         - Hypervisor Emulation Assistance
         - Program - Privileged Instruction
      b. Hypervisor Facility Unavailable
      c. Facility Unavailable
      d. Data Storage for the case of *Fixed-Point Load or Store Caching Inhibited* instructions with MSR_{DR}=1 or the case of a reserved function code for an Atomic Memory Operation
      e. all other Data Storage, Hypervisor Data Storage, [Hypervisor] Data Segment, Machine Check for invalid control memory access, or Alignment (other than for incorrectly aligned prefixed instruction)
      f. Trace

   B. Floating-Point Loads and Stores
      a. Hypervisor Emulation Assistance
      b. Hypervisor Facility Unavailable
      c. Floating-Point Unavailable
      d. [Hypervisor] Data Storage, [Hypervisor] Data Segment, Machine Check for invalid control memory access, or Alignment (other than for incorrectly aligned prefixed instruction)
      e Trace

   C. Vector Loads and Stores
      a. Hypervisor Emulation Assistance
      b. Hypervisor Facility Unavailable
      c. Vector Unavailable
      d. [Hypervisor] Data Storage, [Hypervisor] Data Segment, Machine Check for invalid control memory access, or Alignment (other than for incorrectly aligned prefixed instruction)
      e. Trace

   D. VSX Loads and Stores

a. Hypervisor Emulation Assistance
b. Hypervisor Facility Unavailable
c. VSX Unavailable
d. [Hypervisor] Data Storage, [Hypervisor] Data Segment, Machine Check for invalid control memory access, or Alignment (other than for incorrectly aligned prefixed instruction)
e. Trace

E. Other Floating-Point Instructions
a. Hypervisor Emulation Assistance
b. Hypervisor Facility Unavailable
c. Floating-Point Unavailable
d. Program - Precise Mode Floating-Point Enabled Exception
e. Trace

F. Other Vector Instructions
a. Hypervisor Emulation Assistance
b. Hypervisor Facility Unavailable
c. Vector Unavailable
d. Trace

G. Other VSX Instructions
a. Hypervisor Emulation Assistance
b. Hypervisor Facility Unavailable
c. VSX Unavailable
d. Program - Precise Mode Floating-Point Enabled Exception
e. Trace

H. ***rfebb***, ***rfscv***, ***rfid***, ***hrfid***, ***urfid***, and ***mtmsr***[***d***]
a. These exceptions are mutually exclusive and have the same priority:
- Program - Privileged Instruction, for all except ***rfebb***
- Hypervisor Emulation Assistance, for ***rfebb***, ***rfscv***, ***hrfid*** and ***mtmsr***

b. Hypervisor Facility Unavailable (***rfebb*** only)
c. Facility Unavailable (***rfebb*** only)
d. Program - Floating-Point Enabled Exception, for all except ***rfebb***
e. Trace, for ***rfebb*** and ***mtmsr***[***d***] only

I. Other Instructions
a. These exceptions or groups of exceptions are mutually exclusive and have the same priority (the members of a group are not mutually exclusive, but have the same priority):
- Program - Trap
- System Call
- System Call Vectored
- Hypervisor Emulation Assistance or Program (Privileged Instruction)

b. Hypervisor Facility Unavailable
c. Facility Unavailable
d. Trace

J. [Hypervisor] Instruction Storage, Instruction Segment, Alignment for incorrectly aligned prefixed instruction, and Machine Check for invalid control memory access

These exceptions have the lowest priority in this category. They are recognized only when all instructions prior to the instruction causing one of these exceptions appear to have completed and that instruction is the next instruction to be executed. The four exceptions are mutually exclusive.

The priority of these exceptions is specified for completeness and to ensure that they are not given more favorable treatment. It is acceptable for an implementation to treat these exceptions as though they had a lower priority.

4. Program - Imprecise Mode Floating-Point Enabled Exception

This exception is the fourth highest priority exception. When this exception is created, the interrupt mechanism waits for all other possible exceptions to be reported. It then generates this interrupt if no higher priority exception exists when the interrupt is to be generated.

5. Directed Ultravisor Doorbell

This exception is the fifth highest priority exception. When this exception is created, the interrupt mechanism waits for all other possible exceptions to be reported. It then generates this interrupt if no higher priority exception exists when the interrupt is to be generated.

Programming Note

Some platform implementations may depend on timely servicing of Hypervisor Maintenance interrupts, e.g. to prevent physical damage. The Directed Ultravisor Doorbell interrupt handler may test the HMER to identify such circumstances and take appropriate action.

6. Hypervisor Maintenance

This exception is the sixth highest priority exception. When this exception is created, the interrupt mechanism waits for all other possible exceptions to be reported. It then generates this interrupt if no higher priority exception exists when the interrupt is to be generated.

If a Hypervisor Maintenance exception exists and each attempt to execute an instruction when the Hypervisor Maintenance interrupt is enabled causes an exception (see the Programming Note below), the Hypervisor Maintenance interrupt is not delayed indefinitely.

7. Hypervisor Virtualization, Direct External, Mediated External, and [Hypervisor] Decrementer, Performance Monitor, Directed Privileged Doorbell, Directed Hypervisor Doorbell

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=0 and a Direct External exception exists and each attempt to execute an instruction when this interrupt is enabled causes an exception (see the Programming Note below), the Direct External interrupt is not delayed indefinitely.

Programming Note

An incorrect or malicious operating system could corrupt the first instruction in the interrupt vector location for an instruction-caused interrupt such that the attempt to execute the instruction causes the same exception that caused the interrupt (a looping interrupt; e.g., *Trap* instruction and Program interrupt). Similarly, the first instruction of the interrupt vector for one instruction-caused interrupt could cause a different instruction-caused interrupt, and the first instruction of the interrupt vector for the second instruction-caused interrupt could cause the first instruction-caused interrupt (e.g., Program interrupt and Floating-Point Unavailable interrupt). The looping caused by these and similar cases is terminated by the occurrence of a System Reset or Hypervisor Decrementer interrupt.

7.10 Relationship of Event-Based Branches to Interrupts

7.10.1 EBB Exception Priority

Event-based branches have a priority lower than that of all interrupts. When an event-based exception is created, the Event-Based Branch facility waits for all possible exceptions that would cause interrupts to be reported. It then generates the event-based branch if no exception that would cause an interrupt exists when the event-based branch is to be generated.

7.10.2 EBB Synchronization

When an event-based branch occurs, EBBRR is set to point to an instruction such that all preceding instructions have completed execution, no subsequent instruction has begun execution, and the instruction addressed by EBBRR has not completed execution.

7.10.3 EBB Classes

Event-based branches are classified by whether they are directly caused by the execution of an instruction or are caused by some other system exception. Those that are “system-caused” are

- Performance Monitor
- External

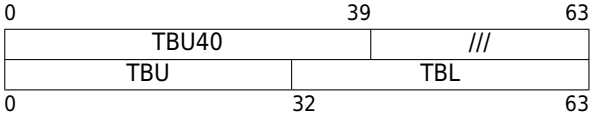
Chapter 8. Timer Facilities

8.1 Overview

The Time Base, Decrementer, Hypervisor Decrementer, Processor Utilization of Resources, and Scaled Processor Utilization of Resources registers provide timing functions for the system. The remainder of this section describes these registers and related facilities.

8.2 Time Base (TB)

The Time Base (TB) is a 64-bit register (see Figure 8.1) containing a 64-bit unsigned integer that is incremented periodically.



Field	Description
TBU40	Upper 40 bits of Time Base
TBU	Upper 32 bits of Time Base
TBL	Lower 32 bits of Time Base

Figure 8.1: 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 8.1. When a ***mtspr*** instruction is executed specifying one of these SPRs, the associated field of the Time Base is altered and the remaining bits of the Time Base are not affected.

See Chapter 5 of Book II for infromation 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.

Programming Note

If software initializes the Time Base on power-on to some reasonable value and the update frequency of the Time Base is constant, the Time Base can be used as a source of values that increase at a constant rate, such as for time stamps in trace entries.

Even if the update frequency is not constant, values read from the Time Base are monotonically increasing (except when the Time Base wraps from $2^{64}-1$ to 0). If a trace entry is recorded each time the update frequency changes, the sequence of Time Base values can be post-processed to become actual time values.

Successive readings of the Time Base may return identical values.

If Time Base bits 60:63 are used as part of a random number generator, software must account for the fact that these bits are set to 0x0 only when bit 59 changes state regardless of whether or not they incremented to 0xF since they were previously set to 0x0.

See the description of the Time Base in Chapter 5 of Book II for ways to compute time of day in POSIX format from the Time Base.

Architecture Note

Disabling the Time Base or making the Time Base SPRs privileged for ***mfspr*** prevents the Time Base from being used to implement a "covert channel" in a secure system.

The potential security exposure above for the Time Base may also apply to other SPRs that measure time and can be read in problem state (e.g., Performance Monitor registers).

Engineering Note

It is intended that the Time Base be useful for timing reasonably short sequences of code (a few hundred instructions) and for low-overhead time stamps for tracing. The Time Base may increment faster or slower than the CPU instruction execution rate; however, it should not increment enormously faster. In most applications it is appropriate for it to increment somewhat slower than the instruction execution rate.

The Time Base driving frequency is also used to update the Decrementer (see Section 8.4) and Hypervisor Decrementer (see Section 8.5), which are used by system software to set interval timers ("alarms"). Additionally, the update frequency of the Processor Utilization of Resources Register (see Section 8.6) is related to the Time Base driving frequency.

8.2.1 Writing the Time Base

Writing the Time Base is privileged, and can be done only in hypervisor state. Reading the Time Base is not privileged; it is discussed in Chapter 5 of Book II.

It is not possible to write the entire 64-bit Time Base using a single instruction. The ***mttbl*** and ***mttbu*** extended mnemonics write the lower and upper halves of the Time Base (TBL and TBU), respectively, preserving the other half. These are extended mnemonics for the ***mtspr*** instruction; Table 5.1.

The Time Base can be written by a sequence such as:

```
lwz     Rx,upper      # load 64-bit value for
lwz     Ry,lower      # TB into Rx and Ry
li      Rz,0
mttbl   Rz            # set TBL to 0
mttbu   Rx            # set TBU
mttbl   Ry            # set TBL
```

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.

```
mftb    Ry            # Read 64-bit Time Base value
clrldi  Ry,Ry,40      # lower 24 bits of old TB
mttbu40 Rx            # write upper 40 bits of TB
mftb    Rz            # read TB value again
clrldi  Rz,Rz,40      # lower 24 bits of new TB
cmpld   Rz,Ry         # compare new and old lwr 24
bge     done          # no carry out of low 24 bits
addis   Rx,Rx,0x0100  # increment upper 40 bits
mttbu40 Rx            # update to adjust for carry
```

Programming Note

The instructions for writing the Time Base are mode-independent. Thus code written to set the Time Base will work correctly in either 64-bit or 32-bit mode.

8.3 Virtual Time Base

The Virtual Time Base (VTB) is a 64-bit incrementing counter.

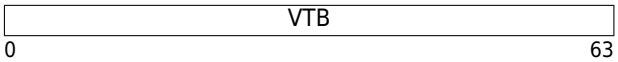


Figure 8.2: Virtual Time Base

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.

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.

8.4 Decrementer

The Decrementer (DEC) is a decrementing counter that provides a mechanism for causing a Decrementer interrupt after a programmable delay.

The Decrementer is driven at the same frequency as the Time Base.

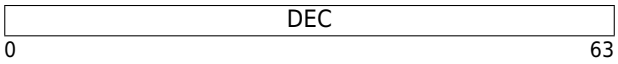


Figure 8.3: Decrementer

The LPCR is used to enable and disable Large Decrementer mode, as defined below. (See Section 2.2.)

When the Decrementer is not in Large Decrementer mode, it behaves as a 32-bit signed integer and operates as follows.

The Decrementer counts down until its value becomes 0x0000_0000_0000_0000; at the next decrement its value becomes 0x0000_0000_FFFF_FFFF. When reading the Decrementer using ***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 0:63-d are ignored by the hardware.

Programming Note

In Large Decrementer mode, the maximum positive value supported by the Decrementer is $2^{d-1}-1$, represented with bits 0:64-d containing 0's and bits 65-d:63 containing 1's. The minimum value supported by the Decrementer is -2^{d-1}, represented with bits 0:64-d containing 1's and bits 65-d:63 containing 0's.

When in Large Decrementer mode, the Decrementer operates as follows.

The binary value of the Decrementer counts down until its value becomes 0x0000_0000_0000_0000; at the next decrement its value becomes 0xFFFF_FFFF_FFFF_FFFF.

When the contents of the DEC_0 change from 0 to 1, a Decrementer exception will come into existence within a reasonable period of time. When the contents of DEC_0 change from 1 to 0, the existing Decrementer exception, if any, will cease to exist within a reasonable period of time, but not later than the completion of the next context synchronizing instruction or event.

The preceding paragraph applies regardless of whether the change in the contents of DEC_0 is the result of decrementation of the Decrementer by the hardware or of modification of the Decrementer caused by execution of an ***mtspr*** instruction.

The operation of the Decrementer has the following additional properties.

1. Loading a GPR from the Decrementer has no effect on the accuracy of the Decrementer.
2. Copying the contents of a GPR to the Decrementer replaces the contents of the 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 Decrementer input frequency will also change. Software must be aware of this in order to set interval timers.

If Decrementer bits 60:63 are used as part of a random number generator, software must account for the fact that these bits are set to 0xF only when bit 59 changes state regardless of whether or not they decremented to 0x0 since they were previously set to 0xF.

8.4.1 Writing and Reading the Decrementer

The contents of the Decrementer can be read or written using the ***mfspr*** and ***mtspr*** instructions, both of which are privileged when they refer to the Decrementer. Using an extended mnemonic (Table 5.1), the Decrementer can be written from GPR Rx using:

```
mtdec  Rx
```

The Decrementer can be read into GPR Rx using:

```
mfdec  Rx
```

Copying the Decrementer to a GPR has no effect on the Decrementer contents or on the interrupt mechanism.

8.5 Hypervisor Decrementer

The Hypervisor Decrementer is a h-bit decrementing counter that is sign-extended to 64 bits. The value of h is implementation dependent, however the number of bits supported by the Hypervisor Decrementer must be greater than or equal to the number of bits supported by the Decrementer. When the Hypervisor 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 with bits 0:64-h containing 1's and bits 65-h:63 containing 0's.

The binary value of the Hypervisor Decrementer counts down until its value becomes 0x0000_0000_0000_0000; at the next decrement its value becomes 0xFFFF_FFFF_FFFF_FFFF.

When the contents of $HDEC_0$ 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 $HDEC_0$ 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 $HDEC_0$ is the result of decrementation of the Hypervisor Decrementer by the hardware or of modification of the Hypervisor Decrementer caused by execution of an ***mtspr*** instruction.

The operation of the Hypervisor Decrementer has the following additional properties.

1. Loading a GPR from the Hypervisor Decrementer has no effect on the accuracy of the Hypervisor Decrementer.
2. Copying the contents of a GPR to the Hypervisor Decrementer replaces the contents of the Hypervisor Decrementer with the contents of the GPR.

Programming Note

In systems that change the Time Base update frequency for purposes such as power management, the Hypervisor Decrementer update frequency will also change. Software must be aware of this in order to set interval timers.

If Hypervisor Decrementer bits 60:63 are used as part of a random number generator, software must account for the fact that these bits are set to 0xF only when bit 59 changes state regardless of whether or not they decremented to 0x0 since they were previously set to 0xF.

Programming Note

A Hypervisor Decrementer exception is not created if the thread is in a power-saving mode when $HDEC_0$ changes from 0 to 1 because having a Hypervisor Decrementer interrupt occur almost immediately after exiting the power-saving mode in this case is deemed unnecessary. The hypervisor already has control, and if a timed exit from the power-saving mode is necessary and possible, the hypervisor can use the Decrementer to exit the power-saving mode at the appropriate time. For some power-saving levels, the state of the Hypervisor Decrementer and Decrementer is not necessarily maintained and updated.

8.6 Processor Utilization of Resources Register (PURR)

The Processor Utilization of Resources Register (PURR) is a 64-bit counter, the contents of which provide an estimate of the resources used by the thread. The contents of the PURR are treated as a 64-bit unsigned integer.

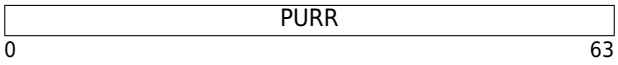


Figure 8.4: Processor Utilization of Resources Register

The PURR is a hypervisor resource; see Chapter 2.

The contents of the PURR increase monotonically, unless altered by software, until the sum of the contents plus the amount by which it is to be increased exceed 0xFFFF_FFFF_FFFF_FFFF ($2^{64} - 1$) at which point the contents are replaced by that sum modulo 2^{64}. There is no interrupt or other indication when this occurs.

The rate at which the value represented by the contents of the PURR increases is an estimate of the portion of resources used by the thread per unit time with respect to other threads that share those resources monitored by the PURR. When the thread is idle, the rate at which the PURR value increases is implementation dependent.

Let the difference between the value represented by the contents of the Time Base at times T_a and T_b be T_{ab}. Let the difference between the value represented by the contents of the PURR at time T_a and T_b be the value P_{ab}. The ratio of P_{ab}/T_{ab} is an estimate of the percentage of shared resources used by the thread during the interval T_{ab}. For the set {S} of threads that share the resources monitored by the PURR, the sum of the usage estimates for all the threads in the set is 1.0.

The definition of the set of threads S, the shared resources corresponding to the set S, and specifics of the algorithm for incrementing the PURR are implementation-specific.

The PURR is implemented such that:

1. Loading a GPR from the PURR has no effect on the accuracy of the PURR.
2. Copying the contents of a GPR to the PURR replaces the contents of the PURR with the contents of the GPR.

Programming Note

Estimates computed as described above may be useful for purposes related to resource utilization, including utilization-based system management and planning.

Because the rate at which the PURR accumulates resource usage estimates is dependent on the frequency at which the Time Base is incremented, and the frequency of the oscillator that drives instruction execution may vary independently from that of the Time Base, the interpretation of the contents of the PURR may be inaccurate as a measurement of capacity consumption for accounting purposes. The SPURR should be used for accounting purposes.

8.7 Scaled Processor Utilization of Resources Register (SPURR)

The Scaled Processor Utilization of Resources Register (SPURR) is a 64-bit counter, the contents of which provide an estimate of the resources used by the thread. The contents of the SPURR are treated as a 64-bit unsigned integer.

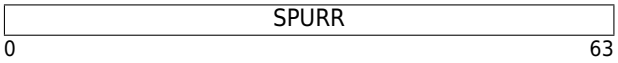


Figure 8.5: Scaled Processor Utilization of Resources Register

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.

8.8 Instruction Counter

The Instruction Counter (IC) is a 64-bit incrementing counter that counts the number of instructions that the thread has completed (according to the sequential execution model; see Section 2.2 of Book I).

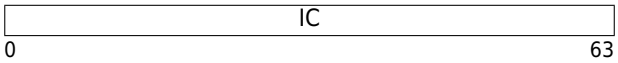


Figure 8.6: Instruction Counter

Chapter 9. Dynamic Execution Control Facility

9.1 Overview

The dynamic execution control facility provides control over certain architectural and micro-architectural aspects of execution especially regarding out-of-order execution behavior. This facility provides a means by which software can request different dynamic execution behaviors in the hardware without the necessity of firmware updates. The facility consists of execution control instructions such as the execution serializing no-op instruction (*ori R31,R31,0*) (see Section 9.2) and three Special Purpose Registers for execution control, the Dynamic Execution Control Register (DEXCR) and the Hypervisor Dynamic Execution Control Register (HDEXCR), and the implementation dependent Ultravisor Dynamic Execution Control Register (UDEXCR) (see Section 9.3). The control provided by these registers is described in Section 9.4.

The ***mfspr*** and ***mtspr*** instructions (see Section 5.4.4) provide access to these registers.

9.2 Dynamic Execution Control Instructions

Instructions described in this Section provides dynamic control over certain architectural and micro-architectural aspects of execution as a direct or side effect of their own execution.

9.2.1 Execution Serializing No-op Instruction

ori R31,R31,0 is a no-op instruction that is also execution serializing: that is, executing an *ori R31,R31,0* instruction ensures that all instructions preceding the *ori R31,R31,0* instruction have completed before the *ori R31,R31,0* instruction completes, and that no subsequent instructions are initiated, even out-of-order, until after the *ori R31,R31,0* instruction completes. The *ori R31,R31,0* instruction may complete before storage accesses associated with instructions preceding the *ori R31,R31,0* instruction have been performed.

Extended Mnemonics:

Additional extended mnemonic for the execution serializing form of *Or Immediate*:

Extended mnemonic:	Equivalent to:
`exser`	`ori 31,31,0`

Programming Note

Warning: Other forms of *ori Rx,Rx,0* that are not described in this section may also have micro-architectural effects on program execution. Use of these forms should be avoided except when software needs the associated micro-architectural effects. If a no-op is needed, the preferred no-op (*ori 0,0,0*) should be used.

Engineering Note

Each side effect that the architecture assigns to a specific form of no-op (e.g. *ori R31,R31,0* described in Section 9.2.1 and *or Rx,Rx,Rx* described in Section 5.4.3) should only be applied to the corresponding instruction form to avoid unintended performance consequences. This principle also governs the application of implementation-specific side effects. As an extreme example, if a no-op that some design implements as group-ending is implemented as execution serializing, performance could be significantly degraded. (Compilers customarily make liberal use of the group-ending no-op on the assumption that other designs will apply no other side effects to the no-op.)

Programming Note

This no-op is intended to be used by software for providing protection against the Spectre class of transient execution attacks by restricting eventually discarded out-of-order execution effects with transient register values, which may compromise confidential data via other covert channels.

9.3 Dynamic Execution Control Registers

The Special Purpose Registers defined for dynamic execution control can control up to 32 unique dynamic execution aspects. The effective aspect value to be used for a particular aspect for each privilege state is defined by the equations in Section 9.4, as is the assignment of

dynamic execution aspects to aspect numbers.

9.3.1 Dynamic Execution Control Register (DEXCR)

The layout of the 64-bit Dynamic Execution Control Register (DEXCR) is shown in Figure 9.1 below. PNH refers to privileged non-hypervisor state. PRO refers to problem state.

PNH0	PNH1	...	PNH31	PRO0	PRO1	...	PRO31
0	1	2	31	32	33	34	63

Bits	Name	Description
0	PNH0	PNH value for aspect number 0
1	PNH1	PNH value for aspect number 1
...	...	...
n	PNH[n]	PNH value for aspect number n
...	...	...
31	PNH31	PNH value for aspect number 31
32	PRO0	PRO value for aspect number 0
33	PRO1	PRO value for aspect number 1
...	...	...
m	PRO[m-32]	PRO value for aspect number m-32
...	...	...
63	PRO31	PRO value for aspect number 31

Figure 9.1: Dynamic Execution Control Register (DEXCR)

This register can be read and written in privileged state using SPR 828. Bits 32:63 of the register can only be read in problem state using SPR 812; ***mfspr*** 812 returns 0s for bits 0:31.

9.3.2 Hypervisor Dynamic Execution Control Register (HDEXCR)

The layout of the 64-bit Hypervisor Dynamic Execution Control Register (HDEXCR) is shown in Figure 9.2 below. HNU refers to hypervisor non-ultravisor state. ENF refers to enforcement of a value to lower privilege states by hypervisor state.

HNU0	HNU1	...	HNU31	ENF0	ENF1	...	ENF31
0	1	2	31	32	33	34	63

Bits	Name	Description
0	HNU0	HNU value for aspect number 0
1	HNU1	HNU value for aspect number 1
...	...	...
n	HNU[n]	HNU value for aspect number n
...	...	...
31	HNU31	HNU value for aspect number 31
32	ENF0	ENF value for aspect number 0
33	ENF1	ENF value for aspect number 1
...	...	...
m	ENF[m-32]	ENF value for aspect number m-32
...	...	...
63	ENF31	ENF value for aspect number 31

Figure 9.2: Hypervisor Dynamic Execution Control Register (HDEXCR)

This register can be read and written in hypervisor state using SPR 471. Bits 32:63 of the register can only be read in non-hypervisor state using SPR 455; ***mfspr*** 455 returns 0s for bits 0:31.

Engineering Note

Note that DEXCR and HDEXCR are both per hardware thread registers and must be swapped on thread reconfiguration like other per thread registers.

9.3.3 Ultravisor Dynamic Execution Control Register (UDEXCR)

The Ultravisor Dynamic Execution Control Register (UDEXCR) is an implementation-dependent register or similar mechanism containing bits ULT0 through ULT31, which are equivalent to HDEXCR bits HNU0 through HNU31 except applying in ultravisor state instead of in hypervisor non-ultravisor state. UDEXCR is set, by an implementation-dependent method, only during system initialization.

The contents of UDEXCR must be the same for all threads under the control of a given instance of the ultravisor; otherwise all results are undefined.

Programming Note

Note that there is no need for ENF bits in UDEXCR, since ultravisor is capable of intercepting entry of execution of a secure partition or process and can set the ENF bits of the HDEXCR register for enforcement purposes as described in Section 9.3. Hence these bits only control the aspects of execution in ultravisor state.

9.4 Dynamic Execution Control Operation

The effective aspect control value for a particular dynamic execution aspect and privilege state is determined by the bits from DEXCR and HDEXCR described in previous section using the following equations. Note that the ENF bit fields in HDEXCR provide a way for the hypervisor or ultravisor to enforce a value of 1 for a particular aspect, but it cannot enforce a value of 0. Hence, the hypervisor or ultravisor can only enforce the behavior determined by positive polarity.

Problem state (MSR_{PR}=1):
effective value for aspect number k = ENF[k] | PRO[k]

Privileged non-hypervisor state ($MSR_{HV\ PR}$=0b00):
effective value for aspect number k = ENF[k] | PNH[k]

Hypervisor non-ultravisor state ($MSR_{S\ HV\ PR}$=0b010):
effective value for aspect number k = HNU[k]

Ultravisor state ($MSR_{S\ HV\ PR}$=0b110):
effective value for aspect number k = ULT[k]

Programming Note

Operating systems should provide system call to allow application programs to request to set PRO[k] bits of DEXCR. Operating system software must save and restore DEXCR on context switches. Likewise, hypervisor software must save and restore HDEXCR when swapping partitions. The hypervisor can enforce certain execution aspects for a partition by setting the ENF[k] bits in HDEXCR. The hypervisor also maintains control of its own execution aspect value via the HNU[k] bits while the operating system maintains control of its own execution aspect value via the PRIV[k] bits unless overriden by the hypervisor.

Programming Note

Changing the value of HNU[k] while in hypervisor non-ultravisor state or changing the value of PNH[k] while in privileged non-hypervisor state may not cause immediate change of execution behavior. See the other programming notes in this section.

The assignment of aspect numbers to dynamic execution aspects in the processor is described below, alongside the interpretation of the effective values for each aspect.

Aspect **Description**

0 **Speculative Branch Hint Enable (SBHE)**

- 0 The hints provided by BO field of Branch instructions may be ignored during speculative execution
- 1 The hints provided by BO field of Branch instructions are obeyed during speculative execution

1:2 **Reserved**

3 **Indirect Branch Recurrent Target Prediction Disable (IBRTPD)**

- 0 Target address prediction mechanisms may be enabled for indirect branches that are not regarded as subroutine returns for prediction purposes (target address is likely to be same as the target address used the preceeding time the branch was taken)
- 1 Target address prediction mechanisms do not record or predict branch target addresses for all indirect branches that are not regarded as subroutine returns for prediction purposes

4 **Subroutine Return Address Prediction Disable (SRAPD)**

- 0 Target address prediction mechanisms may be enabled for indirect branches that are regarded as subroutine returns for prediction purposes
- 1 Target address prediction mechanisms do not record or predict branch target addresses for all indirect branches that are regarded as subroutine returns for prediction purposes

5 **Non-Privileged Hash Instruction Enable (NPHIE)**

- 0 ***hashst*** and ***hashchk*** instructions are executed as no-ops (even when allowed by PCR)
- 1 ***hashst*** and ***hashchk*** instructions are executed normally (if allowed by PCR)

6 **Privileged Hash Instruction Enable (PHIE)**

- 0 ***hashstp*** and ***hashchkp*** instructions are executed as no-ops (even when allowed by PCR)
- 1 ***hashstp*** and ***hashchkp*** instructions are executed normally (if allowed by PCR)

7:31 **Reserved**

Programming Note

In some micro-architectures, the execution behavior controlled by aspect 0 is difficult to change with any degree of timing precision. The change may also bleed over into other threads on the same processor. Any environment that has a dependence on the more secure setting of aspect zero should not change the value, and ideally should share a processor only with similar threads. For other environments, changes to the effective value of aspect 0 represent a relative risk tolerance for its aspect of execution behavior, with the understanding that there will be significant hysteresis in the execution behavior.

Engineering Note

Since aspects 1 and 2 affect only problem state behavior, HNU[1:2], PNH[1:2] and ULT[1:2] have no function.

Programming Note

XL-form *Branch* instructions such as ***bclr***[***l***], ***bcctr***[***l***] and ***bctar***[***l***] are referred to as indirect branches in this chapter. Aspect 3 is intended to control target prediction behavior of ***bclr***[***l***] with BH field value of 0b01, ***bcctr***[***l***] and ***bctar***[***l***] instructions. Aspect 4 is intended to control target prediction behavior of ***bclr***[***l***] with BH field value of 0b00. (See Figure 2.7 in Section 2.4)

Programming Note

Software synchronization is required when changing the effective control aspect value for aspects 3 through 4 to ensure that the execution behavior has changed. See Table 13.2 on page 1279.

Programming Note

In general, aspect values are assigned such that the 1 value may provide greater security. (The 0 value may provide better performance.) This is not strictly enforced.

Engineering Note

Since aspect 6 affects behavior only in privileged states, PRO[6] has no function.

Chapter 10. Debug Facilities

10.1 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 10.2), Completed Instruction Address Breakpoint Register (see Section 10.3), and the Data Address Watchpoint Register (DAWRn) and Data Address Watchpoint Register Extension (DAWRXn) (see Section 10.4). The interrupt associated with the Data Address Breakpoint registers is described in Section 7.5.3. The interrupt associated with the Completed Instruction Address Breakpoint Register is described in Section 7.5.15. The Trace facility, which can be used for single-stepping as well as for control flow tracing, is described in Section 7.5.15.

The ***mfspr*** and ***mtspr*** instructions (see Section 5.4.4) provide access to the registers of the debug facilities.

In addition to the facilities mentioned above, implementations typically provide debug facilities, modes, and access mechanisms that are implementation-specific. For example, implementations typically provide facilities for instruction address tracing, and also access to certain debug facilities via a dedicated interface such as the IEEE 1149.1 Test Access Port (JTAG).

10.2 Come-From Address Register

The Come-From Address Register (CFAR) is a 64-bit register. When an ***rfebb***, ***rfid***, or ***rfscv*** 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.

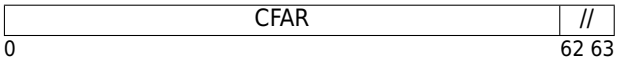


Figure 10.1: Come-From Address Register

The contents of the CFAR can be read and written using the ***mfspr*** and ***mtspr*** instructions. Acccess 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.

Engineering Note

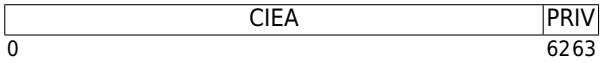
It is preferable to set the CFAR to the exact address of the *Branch* instruction.

10.3 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 10.2, except

that if $SMFCTRL_D$=1 when PRIV≠0, the Privilege specification in the PRIV field is ignored and the facility detects instruction address matches in ultravisor state.



Bit(s)	Name	Description
0:61	CIEA	Completed Instruction Effective Address
62:63	PRIV	Privilege (PRIV > 0b00 ignored when $SMFCTRL_D$=1) 00: Disable matching 01: Match in problem state 10: Match in privileged non-hypervisor state 11: Match in hypervisor non-ultravisor state

Figure 10.2: Completed Instruction Address Breakpoint Register

A Completed Instruction Address Breakpoint match occurs upon instruction completion if all of the following conditions are satisfied. The values of CIABR, SMFCTRL, and the MSR that are used for the comparisons are those that exist at the time the instruction is initiated.

- the completed instruction address is equal to $CIEA_{0:61}$ || 0b00. For prefixed instructions, the completed instruction address is the address of the prefix.
- $SMFCTRL_D$=0 and the thread privilege matches that specified in PRIV or $SMFCTRL_D$=1, PRIV≠0, and $MSR_{S\ HV\ PR}$=0b110.

In 32-bit mode the high-order 32 bits of the EA are treated as zeros for the purpose of detecting a match.

A Completed Instruction Address Breakpoint match causes a Trace exception, which may cause a Trace interrupt as described in Section 7.5.15.

Engineering Note

The initial value of $CIABR_{PRIV}$ (i.e., the value when software is given control during system initialization) must be 0b00.

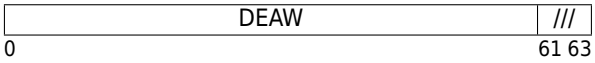
10.4 Data Address Watchpoint

The Data Address Watchpoint mechanism provides a means of detecting load and store accesses to multiple doubleword-aligned effective address (EA) ranges. At least two independent address ranges are provided.

Programming Note

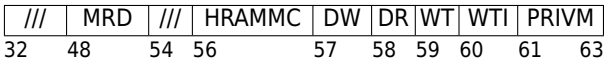
The Data Address Watchpoint mechanism employs a simple EA compare. It makes no attempt to take the radix table translation quadrants (keyed off $EA_{0:1}$) into account to enable a single setting to work in all privilege levels.

Each Data Address Watchpoint range is controlled by a pair of SPRs: [] the Data Address Watchpoint Register (DAWRn), shown in Figure 10.3, and the Data Address Watchpoint Register Extension (DAWRXn), shown in Figure 10.4, where n=0,1,.... $SMFCTRL_D$ functions as an extension to the PRIVM field: when $SMFCTRL_D$=1, the facility detects data address watchpoint matches in ultravisor state in addition to states enabled by the PRIVM field.



Bit(s)	Name	Description
0:60	DEAW	Data Effective Address Watchpoint

Figure 10.3: Data Address Watchpoint Register



Bit(s)	Name	Description
48:53	MRD	Match Range in Doublewords biased by -1. (0b000000 = 1 DW, 0b111111 = 64 DW)
56	HRAMMC	Hypervisor Real Addressing Mode Match Control 0: $DEAW_0$ and EA_0 are used during matching in ultravisor or hypervisor real addressing mode 1: $DEAW_0$ and EA_0 are ignored during matching in ultravisor or hypervisor real addressing mode
57	DW	Data Write
58	DR	Data Read
59	WT	Watchpoint Translation
60	WTI	Watchpoint Translation Ignore
61:63	PRIVM	Privilege Mask
61	HYP	Hypervisor non-ultravisor state
62	PNH	Privileged Non-Hypervisor state
63	PRO	Problem state

All other fields are reserved.

Figure 10.4: Data Address Watchpoint Register Extension

The supported PRIVM values are 0b000, 0b001, 0b010, 0b011, 0b100, and 0b111 when $SMFCTRL_D$=0 and 0b000, 0b001, 0b010, and 0b011 when $SMFCTRL_D$=1. If the combination of $SMFCTRL_D$ and the PRIVM field does not contain one of the supported values, then whether a match occurs for a given storage access is undefined. Elsewhere in this section it is assumed that the PRIVM field contains one of the supported values.

Programming Note

When $SMFCTRL_D$=0, PRIVM value 0b000 causes matches not to occur regardless of the contents of other DAWRn and DAWRXn fields. PRIVM values 0b101 and 0b110 are not supported because a storage location that is shared between the hypervisor and non-hypervisor software is unlikely to be accessed using the same EA by both the hypervisor and the non-hypervisor software. (PRIVM value 0b111 is supported primarily for reasons of software compatibility with respect to emulation of the DABR facility as described in a subsequent Programming Note.)

$SMFCTRL_D$=1 is provided for ultravisor debugging and also for ultravisor supervision of secure partition debugging. When $SMFCTRL_D$=1, exceptions due to matches that occur in hypervisor non-ultravisor state are unlikely to be desirable.

A Data Address Watchpoint match occurs for a *Load* or *Store* instruction, or for an instruction that is treated as a *Load* or *Store*, if, for any byte accessed, all of the following conditions are satisfied.

For the first condition, chk_DEAW and chk_EA are defined as follows.

If $MSR_{HV\ DR}$=0b10 and HRAMMC=1 then
chk_DEAW = 0b0 || $DEAW_{1:60}$ and
chk_EA = 0b0 || $EA_{1:63}$
otherwise
chk_DEAW = DEAW and chk_EA = EA.

- the access is [] located in the range $chk\_DEAW_{0:60} \leq chk\_EA_{0:60} \leq (chk\_DEAW_{0:60} + (^{55}0 \,||\, MRD_{0:5}))$ []
- $(MSR_{DR} = DAWRXn_{WT}) \mid DAWRXn_{WTI}$
- the thread is in
  - ultravisor state and $SMFCTRL_D$=1, or
  - hypervisor non-ultravisor state and $DAWRXn_{HYP}$ = 1, or
  - privileged non-hypervisor state and $DAWRXn_{PNH}$ = 1, or
  - problem state and $DAWRXn_{PR}$ = 1
- the instruction is a *Store* or treated as a *Store* and $DAWRXn_{DW}$ = 1, or the instruction is a *Load* or treated as a *Load* and $DAWRXn_{DR}$ = 1.

In hypervisor and ultravisor real addressing modes, bits 1:63-m of the EA are ignored for the purpose of detecting a match, where m is the real address size supported by the implementation. In virtual real addressing mode, the high order 24 bits of the EA are ignored for the purpose of detecting a match. In 32-bit mode the high-order 32 bits of the EA are treated as zeros for the purpose of detecting a match.

A watched range must not cross any of the following boundaries in their respective circumstances. If it does, the facility will operate correctly with respect to the various control parameters (e.g., PRIVM), but the set of EAs that cause matches is undefined.

- the 2^{64}-byte boundary when HPT translation is being performed in other than virtual real addressing mode (i.e., a range that includes the last and first byte of the 2^{64}-byte effective address space)
- a 2^m-byte boundary when the thread is in hypervisor or ultravisor real addressing mode (i.e., a range that, if the corresponding EAs were used to address storage on a design that ignores the high-order bits of the 60-bit real address that are not supported by the implementation, would include the last and first byte of the 2^m-byte real address space)
- a 2^{62}-byte boundary when Radix Tree translation is being performed (i.e., a quadrant boundary)
- a 2^{40}-byte boundary when HPT translation is being performed in virtual real addressing mode (i.e., a range that, if the corresponding EAs were used to address storage, would include the last and first byte of the reserved virtual segment)
- a 2^{32}-byte boundary when the thread is in 32-bit mode (i.e., a range that, if the corresponding EAs were used to address storage, would include the last and first byte of the 2^{32}-byte effective address space)

If the above conditions are satisfied, it is undefined whether a match occurs in the following cases.

- The instruction is *Store Conditional* but the store is not performed
- The instruction is ***dcbz***. (For the purpose of determining whether a match occurs, ***dcbz*** is treated as a *Store*.)

The *Cache Management* instructions other than ***dcbz*** never cause a match.

A Data Address Watchpoint match causes a Data Storage exception or a Hypervisor Data Storage exception (see Section 7.5.3, “Data Storage Interrupt (DSI)” on page 1207 and Section 7.5.16, “Hypervisor Data Storage Interrupt (HDSI)” on page 1219). 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

The limitations of the Data Address Watchpoint mechanism may lead to uncertainties when attempting to interpret a reported match. For example, there is no explicit indication of which watchpoint matched. The watched address ranges could overlap, for example, as a result of the double-word granularity of the address range specification, making it unclear which watchpoint matched. Even if one of the overlapping watchpoints was set to match only for stores and the other only for loads, the existence of the Atomic Memory Operations would make the reporting of a store in the DSISR inconclusive. For a given watchpoint, the double-word granularity may make the validity of the match uncertain because only the starting address of the match is reported without a length. In such cases, it is necessary to examine the operand that caused the match to determine what actually happened.

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 $DAWRXn_{60:63}$ to 0b0111.

Engineering Note

If a Data Address Watchpoint match occurs, it is preferable not to access any bytes of the storage operand at or after the first matching address. This makes the Data Address Watchpoint mechanism more useful for debugging.

Engineering Note

The initial value of $DAWRXn_{DW\,DR}$ (i.e., the value when software is given control during system initialization) must be 0b00.

Chapter 11. Performance Monitor Facility

11.1 Overview

The Performance Monitor facility provides a means of collecting information about program and system performance.

11.2 Performance Monitor Operation

The Performance Monitor facility includes the following features.

- an MSR bit
  - PMM (Performance Monitor Mark), which can be used to select one or more programs for monitoring
- registers
  - PMC1 - PMC6 (Performance Monitor Counters 1 - 6), which count events
  - MMCR0, MMCR1, MMCR2, MMCR3 and MMCRA (Monitor Mode Control Registers 0, 1, 2, 3 and A), which control the Performance Monitor facility
  - SIAR, SDAR, SIER, SIER2 and SIER3 (Sampled Instruction Address Register, Sampled Data Address Register, Sampled Instruction Event Register, Sampled Instruction Event Register 2 and Sampled Instruction Event Register 3), which contain the address of the "sampled instruction" and of the "sampled data," and additional information about the "sampled instruction" (see Section 11.4.8 – Section 11.4.10).
- 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 11.4.3). A single Performance Monitor alert may reflect multiple enabled conditions and events.
- When a Performance Monitor alert occurs, $MMCR0_{PMAO}$ is set to 1 and the writing of BHRB entries, if in process, is suspended.

  When the contents of $MMCR0_{PMAO}$ change from 0 to 1, a Performance Monitor exception will come into existence within a reasonable period of time. When the contents of $MMCR0_{PMAO}$ 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, $MMCR0_{EBE}$ = 0, and either $HFSCR_{PM}$=1 or the thread is in hypervisor state, an interrupt occurs.
  - If MSR_{PR} = 1, $MMCR0_{EBE}$ = 1, a Performance Monitor event-based exception occurs if $BESCR_{PME}$=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 $BESCR_{GE}$=1.

Programming Note

If a Performance Monitor exception occurs when MSR_{PR}=0 but $MMCR0_{EBE}$ = 1, $BESCR_{PME}$=1, and event-based exceptions are enabled by $FSCR_{EBB}$ and $HFSCR_{EBB}$, the resulting Performance Monitor event-based exception occurs when MSR_{PR} next becomes 1, provided the other bits are set as described above.

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 11.4.4.

11.3 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.

- ***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.

11.4 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 13 “Synchronization Requirements for Context Alterations” on page 1278).

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.

11.4.1 Performance Monitor SPR Numbers

The Performance Monitor registers have two sets of SPR numbers, one set that is non-privileged and another set that is privileged.

For the purpose of explanation elsewhere in the architecture, the non-privileged registers are divided into two groups as defined below.

- A: The non-privileged read/write Performance Monitor registers (i.e., the PMCs, MMCR0, MMCR2, and MMCRA at SPR numbers 771-776, 779, 769, and 770, respectively)
- B: The non-privileged read-only Performance Monitor registers (i.e., SIER2, SIER3, MMCR3, SIER, SIAR, SDAR, and MMCR1 at SPR numbers 736, 737, 738, 768, 780, 781, and 782, respectively).

The SPRs in group B are treated as undefined registers for write (***mtspr***) operations. See the ***mtspr*** instruction description in Section 5.4.4 for additional information.

When the PCR makes a register in either group A or B unavailable in problem state, that SPR is not included in group A or B.

Programming Note

Older versions of Performance Monitor facilities used different sets of SPR numbers from those shown in Section 5.4.4. (All 32-bit PowerPC implementations used a different set.)

Programming Note

The following tables summarize the interrupts that occur as a result of accessing the non-privileged Performance Monitor registers in problem state when $MMCR0_{PMCC}$, 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.)

			mfspr				***mtspr***			
			PMCC				**PMCC**			
	SPR	**#**	**00**	**01**	**10**	**11**	**00**	**01**	**10**	**11**
Group A	MMCR2	769	HU^3	FU, HU^3	HU^3	HU^3	HE,HU^3	FU, HU^3	HU^3	HU^3
	MMCRA	770	HU^3	FU, HU^3	HU^3	HU^3	HE,HU^3	FU, HU^3	HU^3	HU^3
	PMC1	771	HU^3	FU, HU^3	HU^3	HU^3	HE,HU^3	FU, HU^3	HU^3	HU^3
	PMC2	772	HU^3	FU, HU^3	HU^3	HU^3	HE,HU^3	FU, HU^3	HU^3	HU^3
	PMC3	773	HU^3	FU, HU^3	HU^3	HU^3	HE,HU^3	FU, HU^3	HU^3	HU^3
	PMC4	774	HU^3	FU, HU^3	HU^3	HU^3	HE,HU^3	FU, HU^3	HU^3	HU^3
	PMC5	775	HU^3	FU, HU^3	HU^3	FU, HU^3	HE,HU^3	FU, HU^3	HU^3	FU, HU^3
	PMC6	776	HU^3	FU, HU^3	HU^3	FU, HU^3	HE,HU^3	FU, HU^3	HU^3	FU, HU^3
	MMCR0	779	HU^3	FU, HU^3	HU^3	HU^3	HE,HU^3	FU, HU^3	HU^3	HU^3
Group B	SIER2[5]	736	FU^4,HU^3	FU, HU^3	HU^3	HU^3	See 2.	See 2.	See 2.	See 2.
	SIER3[5]	737	FU^4,HU^3	FU, HU^3	HU^3	HU^3	See 2.	See 2.	See 2.	See 2.
	MMCR3[5]	738	FU^4,HU^3	FU, HU^3	HU^3	HU^3	See 2.	See 2.	See 2.	See 2.
	SIER	768	FU^4,HU^3	FU, HU^3	HU^3	HU^3	See 2.	See 2.	See 2.	See 2.
	SIAR	780	FU^4,HU^3	FU, HU^3	HU^3	HU^3	See 2.	See 2.	See 2.	See 2.
	SDAR	781	FU^4,HU^3	FU, HU^3	HU^3	HU^3	See 2.	See 2.	See 2.	See 2.
	MMCR1	782	FU^4,HU^3	FU, HU^3	FU, HU^3	FU, HU^3	See 2.	See 2.	See 2.	See 2.

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 5.4.4 for additional information.) FU or HU interrupts do not occur regardless of the value of $MMCR0_{PMCC}$ or $HFSCR_{PM}$.
3. An HU interrupt occurs if $HFSCR_{PM}$=0 when this SPR is accessed in either problem state or privileged non-hypervisor state.
4. An FU interrupt occurs only if PCR indicates a version of the architecture subsequent to V 3.0. and $MMCR0_{PMCCEXT}$ is set.
5. When the PCR indicates a version of the architecture prior to V 3.1, this SPR is treated as undefined in problem state; no FU or HU interrupts occur regardless of the value of $MMCR0_{PMCC}$ or $HFSCR_{PM}$.

11.4.2 Performance Monitor Counters

The six Performance Monitor Counters, PMC1 through PMC6, are 32-bit registers that count events.

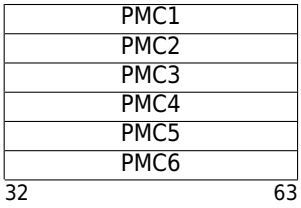


Figure 11.1: Performance Monitor Counter registers

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.

Architecture Note

Because the PMCs count events, they indirectly measure time, and are therefore subject to the same exposures as the Time Base with respect to “covert channels.” (See the first Architecture Note in Section 8.2.) The exposures can be minimized by freezing the counters using $MMCR0_{FC}$ or other “freeze counters” fields in MMCR0 (see Section 11.4.4).

Architecture Note

The PMCs are numbered 1-6, rather than 0-5 which would be more consistent with the numbering in other register names, because early implementations of Performance Monitors numbered them thus.

11.4.2.1 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 $MMCRA_{SE}$=1. The types of instructions that are sampled are specified in $MMCRA_{SM}$ and $MMCRA_{ES}$. Randomly sampled events are specified for each counter by using encodes E0-EF in the PMCn Selector fields in MMCR1.

Programming Note

A typical sequence of operations that enables use the PMCs is as follows.

- Freeze the counters by setting $MMCR0_{FC}$=1.
- Set control fields in MMCR0 and MMCR2 that control counting in various privilege states and other modes, and that enable counter negative conditions.
- Initialize the events to be counted by PMCs 1-4 using the PMCn Selector fields in 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 $MMCR0_{FC}$ to 0 to disable freezing the counters, and set $MMCR0_{PMAE}$ to 1 if a Performance Monitor alert (and the corresponding Performance Monitor interrupt) is desired when an enabled condition or event occurs. (See Section 11.2 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 11.4.4 - 11.4.10 for information regarding MMCRs, SIAR, SDAR, and SIER, and some additional usage examples.

11.4.3 Threshold Event Counter

The threshold event counter and associated controls are in MMCRA (see Section 11.4.7). When Performance Monitor alerts are enabled ($MMCR0_{PMAE}$=1), this counter begins incrementing from value 0 upon each occurrence of the event specified in the Threshold Event Counter Event (TECE) field after the event specified by the Threshold Start Event (TS) field occurs. The counter stops incrementing when the event specified in the Threshold End Event (TE) field occurs. The counter subsequently freezes until the event specified in the TS field is again recognized, at which point it restarts incrementing from value 0 as explained above. If the counter reaches its maximum value or a Performance Monitor alert occurs, incrementing stops. After the Performance Monitor alert occurs, the contents of the threshold event counter are not altered by the hardware until software sets $MMCR0_{PMAE}$ to 1.

Programming Note

Because hardware can modify the contents of the threshold event counter when random sampling is enabled ($MMCRA_{SE}$=1) and $MMCR0_{PMAE}$=1 at any time, any value written to the threshold event counter under this condition may be immediately overwritten by hardware.

The threshold event counter value is represented as a 3-bit integral power of 4, multiplied by an 8-bit integer. The exponent is contained in $MMCRA_{TECX}$, 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 4 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 an 8-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 (255), within 3 events of the exact value for values from 256 - 1020 (or 255×4), within 15 events of the exact value for values from 1024 - 4080 (or 255×4^2), and so on. This represents a relative error in event count of less than 1.6%, 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.

11.4.4 Monitor Mode Control Register 0

Monitor Mode Control Register 0 (MMCR0) is a 64-bit register as shown below.

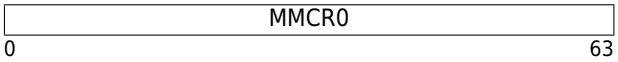


Figure 11.2: Monitor Mode Control Register 0

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 $2 \leq j \leq n$. n=4 when $MMCR0_{PMCC}$=0b11 and n=6 otherwise.

When $MMCR0_{PMCC}$ 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.

Bit(s) Definition

0:31 Reserved

32 **Freeze Counters** (FC)

- 0 The PMCs are incremented (if permitted by other MMCR bits).
- 1 The PMCs are not incremented.

The hardware sets this bit to 1 when an enabled condition or event occurs and $MMCR0_{FCECE}$=1.

33 **Freeze Counters in Privileged State** (FCS)

- 0 The PMCs are incremented (if permitted by other MMCR bits).
- 1 The PMCs are not incremented if $MSR_{HV\ PR}$=0b00.

34 **Conditionally Freeze Counters and BHRB in Problem State** (FCP)

If the value of bit 51 (FCPC) is 0, this field has the following meaning.

- 0 The PMCs are incremented (if permitted by other MMCR bits) and entries are written into the BHRB (if permitted by the BHRB Instruction Filtering Mode field in MMCRA).
- 1 The PMCs are not incremented, and entries are not written into the BHRB, if MSR_{PR}=1.

If the value of bit 51 (FCPC) is 1, this field has the following meaning.

- 0 The PMCs are not incremented, and entries are not written into the BHRB, if $MSR_{HV\ PR}$=0b01.
- 1 The PMCs are not incremented, and entries are not written into the BHRB, if $MSR_{HV\ PR}$=0b11.

Programming Note

In order to freeze counters in problem state regardless of MSR_{HV}, $MMCR0_{FCPC}$ must be set to 0 and $MMCR0_{FCP}$ must be set to 1.

35 **Freeze Counters while Mark = 1** (FCM1)

- 0 The PMCs are incremented (if permitted by other MMCR bits).
- 1 The PMCs are not incremented if MSR_{PMM}=1.

36 **Freeze Counters while Mark = 0** (FCM0)

- 0 The PMCs are incremented (if permitted by other MMCR bits).
- 1 The PMCs are not incremented if MSR_{PMM}=0.

37 **Performance Monitor Alert Enable** (PMAE)

- 0 Performance Monitor alerts are disabled and BHRB entries are not written.
- 1 Performance Monitor alerts are enabled, and BHRB entries are written (if enabled by other bits) until a Performance Monitor alert occurs, at which time:
  - $MMCR0_{PMAE}$ is set to 0
  - $MMCR0_{PMAO}$ is set to 1

Programming Note

Software can set this bit and $MMCR0_{PMAO}$ to 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 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).

38 **Freeze Counters on Enabled Condition or Event** (FCECE)

- 0 The PMCs are incremented (if permitted by other MMCR bits).
- 1 The PMCs are incremented (if permitted by other MMCR bits) until an enabled condition or event occurs when $MMCR0_{TRIGGER}$=0, at which time:
  - $MMCR0_{FC}$ is set to 1

If the enabled condition or event occurs when $MMCR0_{TRIGGER}$=1, the FCECE bit is treated as if it were 0.

39:40 **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

Time Base transition events can be used to collect information about activity, as revealed by event counts in PMCs and by addresses in SIAR and SDAR, at periodic intervals.

In multi-threaded systems in which the Time Base registers are synchronized among the threads, Time Base transition events can be used to correlate the Performance Monitor data obtained by the several threads. For this use, software must specify the same TBSEL value for all the threads in the system.

Because the frequency of the Time Base is implementation-dependent, software should invoke a system service program to obtain the frequency before choosing a value for TBSEL.

41 **Time Base Event Enable** (TBEE)

0 Time Base transition events are disabled.
1 Time Base transition events are enabled.

Programming Note

When PMC3 is configured to count the occurrence of Time Base transition events, the events are counted regardless of the value of $MMCR0_{TBEE}$. (See Section 11.4.5.) The occurrence of a Time Base transition causes a Performance Monitor alert only if $MMCR0_{TBEE}$=1.

42 **BHRB Available** (BHRBA)

This field controls whether the BHRB instructions are available in problem state. If an attempt is made to execute a BHRB instruction in problem state when the BHRB instructions are not available, a Facility Unavailable interrupt will occur.

0 ***clrbhrb*** and ***mfbhrbe*** are not available in problem state.
1 ***clrbhrb*** and ***mfbhrbe*** are available in problem state unless they have been made unavailable by some other register.

43 **Performance Monitor Event-Based Branch Enable** (EBE)

This field controls whether Performance Monitor event-based branches and Performance Monitor event-based exceptions are enabled.

When Performance Monitor event-based branches and exceptions are disabled, no Performance Monitor event-based branches or exceptions occur regardless of the state of $BESCR_{PME}$.

0 Performance Monitor event-based branches and exceptions are disabled.
1 Performance Monitor event-based branches and exceptions are enabled.

Programming Note

In order to enable a problem state application to use the Event-Based Branch facility for Performance Monitor events, privileged software initializes MMCR1 to specify the events to be counted, and sets MMCR2 and MMCRA to specify additional sampling controls. MMCR0 should be initialized with PMCC set to 0b10 or 0b11 (to give problem state access to various Performance Monitor registers), PMAE and PMAO set to 0s (disabling Performance Monitor alerts), and EBE set to 1 (enabling Performance Monitor event-based branches and exceptions to occur). If the Event-Based Branch facility has not been enabled in the FSCR and HFSCR, it must be enabled in these registers as well.

The above operations by the operating system enable the application to control Performance Monitor event-based branching by means of $BESCR_{PME}$ (to enable or disable Performance Monitor event-based branching) and $MMCR0_{PMAE}$ (to enable or disable Performance Monitor alerts).

44:45 **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 11.4.1) 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 5.4.4 for additional information on accessing SPRs that are not supported.

Programming Note

When the PCR makes SPRs unavailable in problem state, they are treated as undefined in problem state, and they are not included in groups A or B regardless of the value of PMCC. [] Accesses to such SPRs in problem state result in Hypervisor Emulation Assistance interrupts regardless of the value of PMCC, and [Hypervisor] Facility Unavailable interrupts do not occur for them. See Section 2.5 for additional information.

00 PMCs 5 - 6 are included in the Performance Monitor.
Group A is read-only, and group B read access behavior is conditional on $MMCR0_{PMCCEXT}$ in problem state. If an attempt is made to write to an SPR in group A in problem state, a Hypervisor Emulation Assistance interrupt will occur.

01 PMCs 5 - 6 are included in the Performance Monitor.
Group A is not allowed to be read or written in problem state, and group B is not allowed to be read in problem state. If an attempt is made, in problem state, to read or write to an SPR in group A, or to read from an SPR in group B, a Facility Unavailable interrupt will occur.

10 PMCs 5 - 6 are included in the Performance Monitor.
Group A is allowed to be read and written in problem state, and group B except for MMCR1 (SPR 782) is allowed to be read in problem state. If an attempt is made to read MMCR1 in problem state, a Facility Unavailable interrupt will occur.

11 PMCs 5 - 6 are not included in the Performance Monitor. See Section 11.4.2 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 7.2.13).

Programming Note

[]
A write operation to a Performance Monitor register in group A that is attempted in problem state when PMCC=0b00 was defined to cause a Hypervisor Emulation Assistance interrupt in order to maintain compatibility with V 2.06. For other values of PMCC, and for PMCC=0b00 when PMCCEXT=1, write or read operations to group A and read operations from group B that are not allowed result in Facility Unavailable interrupts. Facility Unavailable interrupts provide the operating system with more information about the type of disallowed access that was attempted than the Hypervisor Emulation Assistance interrupt provides. See Section 7.2.12 for additional information.

Programming Note

In order to prevent applications from accessing Performance Monitor registers, PMCC is set to 0b01.

In order to allow applications limited control over the Performance Monitor, PMCC is set to 0b10 or 0b11. These values are also used when Performance Monitor event-based branches are enabled.

Programming Note

When PMCs 5 and 6 are not part of the Performance Monitor (i.e., when PMCC = 0b11), they are not controlled by bits in MMCRs, and counter negative conditions in PMCs 5 and 6 do not result in Performance Monitor alerts or exceptions (see Section 11.2) and do not result in Performance Monitor interrupts.

46:47 Reserved

Architecture Note

Bits 46:47 will be among the last to be assigned a meaning. They were the FCTS (Freeze Counters in Transactional State) and FCNTS (Freeze Counters in Non-Transactional State) bits in earlier versions of the architecture.

48 **PMC1 Condition Enable** (PMC1CE)

This bit controls whether counter negative conditions due to a negative value in PMC1 are enabled.

0 Counter negative conditions for PMC1 are disabled.
1 Counter negative conditions for PMC1 are enabled.

49 **PMCj Condition Enable** (PMCjCE)

This bit controls whether counter negative conditions due to a negative value in any PMCj (i.e., in any PMC except PMC1) are enabled.

0 Counter negative conditions for all PMCjs are disabled.
1 Counter negative conditions for all PMCjs are enabled.

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)
- $MMCR0_{TRIGGER}$ is set to 0

See the description of the FCECE bit, above, regarding the interaction between TRIGGER and FCECE.

Programming Note

Uses of TRIGGER include the following.
- Resume counting in the PMCjs when PMC1 becomes negative, without causing a Performance Monitor interrupt. Then freeze all PMCs (and optionally cause a Performance Monitor interrupt) when a PMCj becomes negative. The PMCjs then reflect the events that occurred between the time PMC1 became negative and the time a PMCj becomes negative. This use requires the following MMCR0 bit settings.
  - TRIGGER=1
  - PMC1CE=0
  - PMCjCE=1
  - TBEE=0
  - FCECE=1
  - PMAE=1 (if a Performance Monitor interrupt is desired)
- Resume counting in the PMCjs when PMC1 becomes negative, and cause a Performance Monitor interrupt without freezing any PMCs. The PMCjs then reflect the events that occurred between the time PMC1 became negative and the time the interrupt handler reads them. This use requires the following MMCR0 bit settings.
  - TRIGGER=1
  - PMC1CE=1
  - TBEE=0
  - FCECE=0
  - PMAE=1

51 **Freeze Counters and BHRB in Problem State Condition** (FCPC)

This bit controls the meaning of bit 34 (FCP). See the definition of bit 34 for details.

Programming Note

In order to enable the FCP bit to freeze counters in problem state regardless of MSR_{HV}, $MMCR0_{FCPC}$ must be set to 0.

52 Reserved

53 **Freeze Counters, sampling and threshold counting in Ultravisor State** (FCU)
0 The PMCs are incremented (if permitted by other MMCR bits).
1 The PMCs are not incremented if $MSR_{S\ HV\ PR}$=0b110.

54 **PMCC Extended** (PMCCEXT)
0 If $MMCR0_{PMCC}$ = b00, a SPR in group B is allowed to be read in problem state.
1 If $MMCR0_{PMCC}$ = b00 and an attempt is made to read from an SPR in group B, a Facility Unavailable Interrupt will occur.

55 **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).

56 **Performance Monitor Alert Occurred** (PMAO)
0 A Performance Monitor alert has not occurred since the last time software set this bit to 0.
1 A Performance Monitor alert has occurred since the last time software set this bit to 0.

This bit is set to 1 by the hardware when a Performance Monitor alert occurs. This bit can be set to 0 only by the ***mtspr*** instruction.

Programming Note

Software can set this bit to 1 and set PMAE to 0 to simulate the occurrence of a Performance Monitor alert.

Software should set this bit to 0 after handling the Performance Monitor alert.

57 Reserved

Architecture Note

Bit 57 will be among the last to be assigned a meaning. It was the FCSS (Freeze Counters in Suspended State) bit in earlier versions of the architecture.

58 **Freeze Counters 1-4** (FC1-4)

0 PMC1 - PMC4 are incremented (if permitted by other MMCR bits).
1 PMC1 - PMC4 are not incremented.

59 **Freeze Counters 5–6** (FC5–6)
0 PMC5 - PMC6 are incremented (if permitted by other MMCR bits).
1 PMC5 - PMC6 are not incremented.

60:61 Reserved

62 **Freeze Counters 1–4 in Wait State** (FC1–4WAIT)
0 PMCs 1-4 are incremented (if permitted by other MMCR bits).
1 PMCs 1-4, except for PMCs counting events that are not controlled by this bit, are not incremented if $CTRL_{RUN}$=0.

Programming Note

When PMC 1 is counting cycles, it is not controlled by this bit. See the description of the F0 event in Section 11.4.5.

63 **Freeze Counters in Hypervisor State** (FCH)
0 The PMCs are incremented (if permitted by other MMCR bits).
1 The PMCs are not incremented if $MSR_{S\ HV\ PR}$=0b010.

11.4.5 Monitor Mode Control Register 1

Monitor Mode Control Register 1 (MMCR1) is a 64-bit register as shown below.

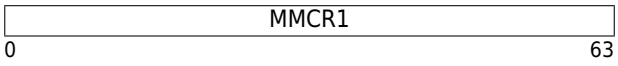


Figure 11.3: Monitor Mode Control Register 1

MMCR1 enables software to specify the events that are counted by the PMCs.

In the following descriptions, events due to randomly sampled instructions occur only if random sampling is enabled ($MMCRA_{SE}$=1); all other events occur whenever the event specification is met regardless of the value of $MMCRA_{SE}$.

Various events defined below refer to “threshold A” through “threshold H”. The table below specifies the number of threshold event counter events corresponding to each of these thresholds.

Threshold	Events
A	4096
B	32
C	64
D	128
E	256
F	512
G	1024
H	2048

Table 11.1: Event Counts for thesholds A–H

Architecture Note

In a future revision of the architecture, additional columns may be added to the above table that allow software to define alternative numbers of events that correspond to each of the above thresholds.

The bit definitions of MMCR1 are as follows. Implementation-dependent MMCR1 bits that are not supported are treated as reserved.

Bit(s) Definition

0:31 Problem state access (SPR 782): Reserved
Privileged access (SPR 782 or 798): Implementation-dependent

32:39 **PMC1 Selector** (PMC1SEL)
The value of PMC1SEL specifies the event to be counted by PMC1 as defined below. All values in the range of E0 - FF that are not specified below are reserved.

Hex	Event
00	Disable events. (No events occur.)
01–BF	Implementation-dependent
C0–DF	Reserved

The following events can occur only when random sampling is enabled ($MMCRA_{SE}$=1). The sampling modes corresponding to each event are listed in parentheses. (The sampling mode is specified in $MMCRA_{SM}$.)

E0 The thread has dispatched a randomly sampled instruction. (RIS)
E2 The thread has completed a randomly sampled *Branch* instruction for which the branch was taken. (RIS, RBS)
E4 The thread has failed to locate a randomly sampled instruction in the primary instruction cache. (RIS)
E6 The threshold event counter has exceeded the number of events corresponding to threshold A (see Table 11.1). (RIS, RLS, RBS)
E8 The threshold event counter has exceeded the number of events corresponding to threshold E (see Table 11.1). (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 *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.

Hex	Event
00	Disable events. (No events occur.)
01–BF	Implementation-dependent
C0–DF	Reserved

The following events can occur only when random sampling is enabled ($MMCRA_{SE}$=1). The sampling modes corresponding to each event are listed in parentheses. (The sampling mode is specified in $MMCRA_{SM}$.)

E0 The thread has obtained the data for a randomly sampled *Load* instruction from storage that did not reside in any cache. (RIS, RLS)

E2 The thread has failed to locate the data for a randomly sampled *Load* instruction in the primary data cache. (RIS, RLS)

E4 The thread filled a block in the primary data cache with data that were accessed by a randomly sampled *Load* instruction and obtained from a location other than the secondary or tertiary cache. (RIS, RLS)

E6 The threshold event counter has exceeded the number of events corresponding to threshold B (see Table 11.1). (RIS, RLS, RBS)

E6 The threshold event counter has exceeded the number of events corresponding to threshold F (see Table 11.1). (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 **PMC3Selector** (PMC3SEL)

The value of PMC3SEL specifies the event to be counted by PMC3 as defined below. All values in the range of E0 - FF that are not specified below are reserved.

Hex	Event
00	Disable events. (No events occur.)
01–BF	Implementation-dependent
C0–DF	Reserved

The following events can occur only when random sampling is enabled ($MMCRA_{SE}$=1). The sampling modes corresponding to each event are listed in parentheses. (The sampling mode is specified in $MMCRA_{SM}$.)

E2 The thread has completed a randomly sampled *Store* instruction to the point at which it has reported all exceptions it will cause. (RIS,RLS)

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 11.1). (RIS, RLS, RBS)

EA The threshold event counter has exceeded the number of events corresponding to threshold G (see Table 11.1). (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 $MMCR0_{TBEE}$.

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.

Hex	Event
00	Disable events. (No events occur.)
01–BF	Implementation-dependent
C0–DF	Reserved

The following events can occur only when random sampling is enabled ($MMCRA_{SE}$=1). The sampling modes corresponding to each event are listed in parentheses. (The sampling mode is specified in $MMCRA_{SM}$.)

E0 The thread has 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 11.1). (RIS, RLS, RBS)

EC The threshold event counter has exceeded the number of events corresponding to threshold H (see Table 11.1). (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.

Architecture 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 MMCRs may be defined to cover the additional selectors.

11.4.6 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.

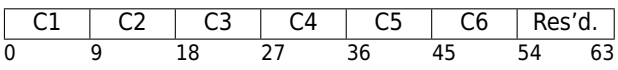


Figure 11.4: Monitor Mode Control Register 2

When $MMCR0_{PMCC}$ = 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 $MMCR0_{PMCC}$ 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 ***mtspr*** is executed in problem state, and all other bits return 0s when ***mfspr*** is executed in problem state.

Bit(s) Definition

0 **Freeze Counter n in Privileged State** (FCnS)
- 0 PMCn is incremented (if permitted by other MMCR bits).
- 1 PMCn is not incremented if $MSR_{HV\ PR}$=0b00.

1 **Freeze Counter n in Problem State if MSR_{HV}=0** (FCnP0)
- 0 PMCn is incremented (if permitted by other MMCR bits).
- 1 PMCn is not incremented if $MSR_{HV\ PR}$=0b01.

Programming Note

Problem state programs need access to this field in order to enable them to individually enable counters when analyzing sections of code. All the other fields will typically be initialized by the operating system.

2 **Freeze Counter n in Problem State if MSR_{HV}=1** (FCnP1)
- 0 PMCn is incremented (if permitted by other MMCR bits).
- 1 PMCn is not incremented if $MSR_{HV\ PR}$=0b11.

3 **Freeze Counter n while Mark = 1** (FCnM1)
- 0 PMCn is incremented (if permitted by other MMCR bits).
- 1 PMCn is not incremented if MSR_{PMM}=1.

4 **Freeze Counter n while Mark = 0** (FCnM0)
- 0 PMCn is incremented (if permitted by other MMCR bits).
- 1 PMCn is not incremented if MSR_{PMM}=0.

5 **Freeze Counter n in Wait State** (FCnWAIT)
- 0 PMCn is incremented (if permitted by other MMCR bits).
- 1 PMCn is not incremented if $CTRL_{RUN}$=0.

Programming Note

The operating system is expected to set $CTRL_{RUN}$ to 0 when the thread is in a “wait state”, i.e., when there is no process ready to run.

6 **Freeze Counter n in Hypervisor State** (FCnH)
- 0 PMCn is incremented (if permitted by other MMCR bits).
- 1 PMCn is not incremented if $MSR_{HV\ PR}$=0b10.

Bits 54:63 of MMCR2 are reserved.

Architecture Note

Bits 54:63 are used to control an implementation-dependent facility in the POWER8 processor. These bits will not be assigned a meaning in versions of the architecture subsequent to V. 2.07 except after careful consideration of the effect of such assignment on POWER8.

11.4.7 Monitor Mode Control Register A

Monitor Mode Control Register A (MMCRA) is a 64-bit register as shown below.

Figure 11.5: Monitor Mode Control Register A

MMCRA gives privileged programs the ability to control the sampling process, BHRB filtering, and threshold events.

When $MMCR0_{PMCC}$ is set to 0b10 or 0b11, providing problem state programs read/write access to MMCRA, the Threshold Event Counter Exponent (TECX) and Threshold Event Counter Multiplier (TECM) fields are read-only, and all other fields return 0s, when ***mfspr*** is executed in problem state; all fields are not changed when ***mtspr*** is executed in problem state.

Programming Note

Read/write access is provided to MMCRA in problem state (SPR 770) when $MMCR0_{PMCC}$ = 0b10 or 0b11 even though no fields can be modified by ***mtspr*** because future versions of the architecture may allow various fields of MMCRA to be modified in problem state.

The bit definitions of MMCRA are as follows.

Bit(s) Definition

0:25 Problem state access (SPR 770): Reserved
Privileged access (SPR 770 or 786): Implementation-dependent

26 **BHRB Recording Disable** (BHRBRD)
This field controls whether BHRB entries are written when BHRB recording is enabled by other bits.
- 0 BHRB entries are written (if enabled by other bits).
- 1 BHRB entries are not written.

27:31 Problem state access (SPR 770): Reserved
Privileged access (SPR 770 or 786): Implementation-dependent

32:33 **BHRB Instruction Filtering Mode** (IFM)
This field controls the filter criterion used by the hardware when recording *Branch* instructions into the BHRB. See Section 11.5.

00 All taken Branch instructions are entered into the BHRB unless prevented by other filtering fields.

01 Only taken calls are entered into the BHRB. Includes direct and indirect: ***bl***, ***bla***, ***bcl***, ***bcla***, ***bclrl***, ***bcctrl***, ***bctarl***.

10 Only taken indirect branches (excluding calls) are entered into BHRB: ***bclr***, ***bcctr***, ***bctar***. This translates to all XL-form taken branches with LK=0.

11 Only taken conditional branches: ***bc***, ***bca***, ***bcl***, ***bcla***, ***bclr***, ***bclrl***, ***bcctr***, ***bcctrl***, ***bctar***, ***bctarl***. This translates to all B-form or XL-form taken branches except those with the BO field encoded to branch always (BO=0b1z1zz).

Programming Note

Filtering mode 10 provides additional filtering for unconditional *Branch* instructions, and for indirect *Branch* instructions only the target address is recorded.

Filtering mode 11 provides additional filtering for instructions that provide a hint or for which the outcome does not depend on the value of the Condition Register.

34:36 **Threshold Event Counter Exponent** (TECX)

This field species the exponent of the threshold event counter value. See Section 11.4.3 for additional information. The maximum exponent supported is at least 5.

37:44 **Threshold Event Counter Multiplier** (TECM)

This field species the multiplier of the threshold event counter value. See Section 11.4.3 for additional information.

Programming Note

When $MMCR0_{PMCC}$ = 0b10 or 0b11, providing problem-state programs read-write access to MMCRA, problem state programs are able to read only the TECX and TECM fields (and are not able to write any fields). The values of these fields are needed during the processing of an event-based branch that occurs due to a counter negative condition for a PMC that was counting “threshold counter exceeded n” events (e.g. $MMCR1_{PMC1SEL}$ = 0xE8). Reading these fields enables the application to determine the amount by which the threshold was exceeded. Applications are not given access to other fields, and these other fields must be initialized by the operating system.

45:47 **Threshold Event Counter Event** (TECE)

This field specifies the event, if any, that is counted by the threshold event counter. The values and meanings are follows.

Value	Event
000	Disable counting.
001	A cycle has occurred.
010	An instruction has completed.
011	Reserved

All other values are implementation-dependent.

48:51 **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 $MMCRA_{SE}$=1 (random sampling enabled) and one of the sampling modes listed in parenthesis is in effect. (The sampling mode that is currently in effect is specified in $MMCRA_{SM}$.)

0000 Reserved.

0001 The thread has randomly sampled an instruction while it is being decoded. (RIS)

0010 The thread has dispatched a randomly sampled instruction. (RIS)

0011 A randomly sampled instruction has been sent to a facility (e.g. *Branch, Fixed Point*, etc.) (RIS, RLS, RBS)

0100 The thread has completed a randomly sampled instruction to the point at which it has reported all exceptions it will cause. (RIS, RLS, RBS)

0101 The thread has completed a randomly sampled instruction. (RIS, RLS, RBS)

0110 The thread has failed to locate data for a randomly sampled *Load* instruction in the primary data cache. (RIS, RLS)

0111 The thread has filled a block in the primary data cache with data that were accessed by a randomly sampled *Load* instruction. (RIS, RLS)

The definition of the following values depends on whether the access to MMCRA is in problem state or in privileged state.

Problem state access (SPR 770)
1000 – 1111 - Reserved

Privileged access (SPR 770 or 786)
1000 – 1111 - Implementation-dependent

Engineering Note

Since all of the values with bit 48=1 are reserved for problem state access, ***mfspr*** RT,770 must always return 0 for bit 48 regardless of the value that privileged programs last wrote to this bit.

52:55 **Threshold End Event** (TE)

This field specifies the event that causes the threshold event counter to stop counting occurrences of the event specified in the Threshold Event Counter Event (TECE) field. The events only occur if $MMCRA_{SE}$=1 (random sampling enabled) and one of the sampling modes listed in paren-

thesis 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

Engineering Note

Since all of the values with bit 52=1 are reserved for problem state access, ***mfspr*** RT,770 must always return 0 for bit 52 regardless of the value that privileged programs last wrote to this bit.

56 Reserved

57:59 **Eligibility for Random Sampling** (ES)

When random sampling is enabled ($MMCRA_{SE}$=1) and the SM field indicates random instruction sampling (RIS), the encodings of this field specify the instructions that are eligible to be sampled as follows.

000 All instructions
001 All *Load* and *Store* instructions
010 All probe no-op instructions
011 Reserved

The definition of the following values depends on whether the access to MMCRA is in problem state or in privileged state.

Problem state access (SPR 770)
100 – 111 - Reserved

Privileged access (SPR 770 or 786)
100 – 111 - Implementation-dependent

When random sampling is enabled ($MMCRA_{SE}$=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 ($MMCRA_{SE}$=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

Engineering Note

Since all of the values with bit 57=1 are reserved for problem state access, ***mfspr*** RT, 770 must always return 0 for bit 57 regardless of the value that privileged programs last wrote to this bit.

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 11.4.2.1 for information about random sampling.

11.4.8 Sampled Instruction Address Register

The Sampled Instruction Address Register (SIAR) is a 64-bit register.

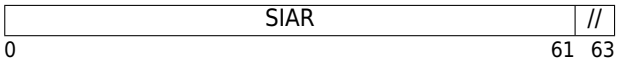


Figure 11.6: Sampled Instruction Address Register

When a Performance Monitor alert occurs because of an event caused by execution of a randomly sampled instruction, bits 0:61 of the SIAR contain bits 0:61 of the effective address of the instruction if $SIER_{SIARV}$ = 1 and contain an undefined value if $SIER_{SIARV}$ = 0.

When a Performance Monitor alert occurs because of an event other than an event caused by execution of a randomly sampled instruction, the SIAR contains the effective address of an instruction that was being executed, possibly out-of-order, at or around the time that the Performance Monitor alert occurred.

The instruction located at the effective address contained in the SIAR is called the “sampled instruction”.

Except as described in the next paragraph, the contents of SIAR may be altered by the hardware if and only if $MMCR0_{PMAE}$=1. Thus after the Performance Monitor alert occurs, the contents of SIAR are not altered by the hardware until software sets $MMCR0_{PMAE}$ to 1. After software sets $MMCR0_{PMAE}$ to 1, the contents of SIAR are undefined until the next Performance Monitor alert occurs.

If single step or branch tracing is active (MSR_{TE} = 0b10 or 0b01), the contents of SIAR as used by the Performance Monitor facility are undefined and may change even when $MMCR0_{PMAE}$=0.

Programming Note

When the Performance Monitor alert occurs, $SIER_{SAMPHV\ SAMPPR}$ indicates the value of $MSR_{HV\ PR}$ that was in effect when the sampled instruction was being executed. (The contents of these SIER bits are visible only in privileged state.)

Engineering Note

If the Performance Monitor alert is caused by an enabled counter negative condition that can be associated with the execution of a specific instruction, it is preferable to set SIAR to that instruction’s address.

11.4.9 Sampled Data Address Register

The Sampled Data Address Register (SDAR) is a 64-bit register.

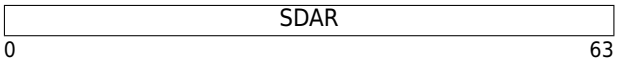


Figure 11.7: Sampled Data Address Register

When a Performance Monitor alert occurs because of an event caused by execution of a randomly sampled instruction, the SDAR contains the effective address of the storage operand of the instruction if $SIER_{SDARV}$ = 1 and contains an undefined value if $SIER_{SDARV}$ = 0.

When a Performance Monitor alert occurs because of an event other than an event caused by execution of a randomly sampled instruction, the SDAR contains the effective address of the storage operand of an instruction that was being executed, possibly out-of-order, at or around the time that the Performance Monitor alert occurred. This storage operand may or may not be the storage operand (if any) of the sampled instruction.

The data located at the effective address contained in the SDAR are called the “sampled data.”

Except as described in the next paragraph, the contents of SDAR may be altered by the hardware if and only if $MMCR0_{PMAE}$=1. Thus after the Performance Monitor alert occurs, the contents of SDAR are not altered by the hardware until software sets $MMCR0_{PMAE}$ to 1. After software sets $MMCR0_{PMAE}$ to 1, the contents of SDAR are undefined until the next Performance Monitor alert occurs.

If single step or branch tracing is active (MSR_{TE} = 0b10 or 0b01), the contents of SDAR as used by the Performance Monitor facility are undefined and may change even when $MMCR0_{PMAE}$=0.

Engineering Note

If the sampled instruction has a storage operand, it is preferable to set SDAR to that storage operand’s address.

11.4.10 Sampled Instruction Event Register

The Sampled Instruction Event Register (SIER) is a 64-bit register.

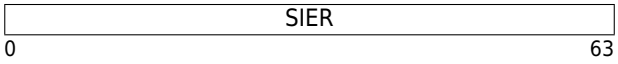


Figure 11.8: Sampled Instruction Event Register

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.

Programming Note

A Performance Monitor alert occurs because of an event caused by execution of a randomly sampled instruction if random sampling is enabled and a counter negative condition exists in a PMC that was counting events based on randomly sampled instructions.

When random sampling is disabled or when a Performance Monitor alert occurs because of an event that was not caused by execution of a randomly sampled instruction, the contents of the SIER are undefined.

The contents of SIER may be altered by the hardware if and only if $MMCR0_{PMAE}$=1. Thus after the Performance Monitor alert occurs, the contents of SIER are not altered by the hardware until software sets $MMCR0_{PMAE}$ to 1. After software sets $MMCR0_{PMAE}$ to 1, the contents of SIER are undefined until the next Performance Monitor alert occurs.

The bit definitions of the SIER are as follows.

Bit(s) Definition

0:37 The definition of these bits depends on whether the access to SIER is in problem state or in privileged state.

Problem state access (SPR 768): Reserved

Privileged access (SPR 768 or 784): Implementation-dependent

38:40 The definition of these bits depends on whether the access to SIER is in problem state or in privileged state.

Problem state access (SPR 768): Reserved

Privileged access (SPR 768 or 784):

38 **Sampled MSR_{PR}** (SAMPPR)
Value of MSR_{PR} when the Performance Monitor alert occurred.

39 **Sampled MSR_{HV}** (SAMPHV)
Value of MSR_{HV} when the Performance Monitor alert occurred.

40 Reserved

41 **SIAR Valid** (SIARV)
Set to 1 when the contents of the SIAR are valid (i.e., they contain the effective address of the sampled instruction); otherwise set to 0.

42 **SDAR Valid** (SDARV)
Set to 1 when the contents of the SDAR are valid (i.e., they contain the effective address of the sampled instruction); otherwise set to 0.

43 **Threshold Exceeded** (TE)
Set to 1 by the hardware if the contents of the threshold event counter exceeded the maximum value when the Performance Monitor alert occurred; otherwise set to 0 by the hardware.

44 **Slew Down**
Set to 1 by the hardware if the processor clock was lower than nominal when the Performance Monitor alert occurred; otherwise set to 0 by the hardware.

45 **Slew Up**
Set to 1 by the hardware if the processor clock was higher than nominal when the Performance Monitor alert occurred; otherwise set to 0 by the hardware.

46:48 **Sampled Instruction Type** (SITYPE)
This field indicates the sampled instruction type. The values and their meanings are as follows.

- 000 The hardware is unable to indicate the sampled instruction type
- 001 *Load* Instruction
- 010 *Store* instruction
- 011 *Branch* Instruction
- 100 *Floating-Point* Instruction other than a *Load* or *Store* instruction
- 101 *Fixed-Point* Instruction other than a *Load* or *Store* instruction
- 110 *Condition Register* or *System Call* instruction
- 111 *Load And Reserve* or *Store Conditional* Instruction

49:51 **Sampled Instruction Cache Information** (SICACHE)
This field provides cache-related information about the sampled instruction.

- 000 The hardware is unable to provide any cache-related information for the sampled insttuction.
- 001 The thread obtained the instruction in the primary instruction cache.
- 010 The thread obtained the instruction in the secondary cache.
- 011 The thread obtained the instruction in the tertiary cache.
- 100 The thread failed to obtain the instruction in the primary, secondary, or tertiary cache
- 101 Reserved
- 110 Reserved
- 111 Reserved

52 **Sampled Instruction Taken Branch** (SITAKBR)

Set to 1 if the SITYPE field indicates a *Branch* instruction and the branch was taken; otherwise set to 0.

53 **Sampled Instruction Mispredicted Branch** (SIMISPRED)

Set to 1 if the SITYPE field indicates a *Branch* instruction and the thread has mispredicted either whether or not the branch would be taken, or if taken, the target address; otherwise set to 0.

54:55 **Sampled Branch Instruction Misprediction Information** (SIMISPREDI)

If SIMISPRED=1, this field indicates how the thread mispredicted the outcome of a *Branch* instruction; otherwise this field is set to 0s.

- 00 The instruction was not a mispredicted *Branch* instruction.
- 01 The thread mispredicted whether or not the branch would be taken because the contents of the Condition Register differed from the predicted contents.
- 10 The thread mispredicted the target address of the instruction.
- 11 Reserved

56 **Sampled Instruction Data ERAT Miss** (SIDERAT)

When the SITYPE field indicates a *Load* or *Store* instruction, this field is set to 1 if the thread has failed to locate an ERAT entry during data address translation for the sampled instruction and otherwise is set to 0.

When the SITYPE field does not indicate a *Load* or *Store* instruction, the contents of this field are undefined.

57:59 **Sampled Instruction Data Address Translation Information** (SIDAXLATE)

This field contains information about data address translation for the sampled instruction. If multiple data address translations were performed, the information pertains to the last translation. The values and their meanings are as follows.

- 000 The instruction did not require data address translation.
- 001 The thread translated the data virtual address using the TLB.
- 010 A PTEG required for data address translation for the instruction was obtained from the secondary cache.
- 011 A PTEG required for data address translation for the instruction was obtained from the tertiary cache.
- 100 A PTEG required for data address translation for the instruction was obtained from storage that did not reside in any cache.
- 101 A PTEG required for data address translation for the instruction was obtained from a cache on a different multi-threaded processor that resides on the same chip as the thread.
- 110 A PTEG required for data address translation for the instruction was obtained from a cache on a different chip from the thread.
- 111 Reserved

Architecture Note

There is no encode corresponding to the primary cache due the significant hardware cost required to support it.

60:62 **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.

- 000 The instruction did not require data address translation.
- 001 The instruction was a *Read* for which the thread obtained the referenced data from the primary data cache.
- 010 The instruction was a *Read* for which the thread obtained the referenced data from the secondary cache.
- 011 The instruction was a *Read* for which the thread obtained the referenced datafrom the tertiary cache.
- 100 The instruction was a *Read* for which the thread obtained the referenced datafrom storage that did not reside in any cache.
- 101 The instruction was a *Read* for which the thread obtained the referenced data from a cache on a different multi-threaded processor that resides on the same chip as the thread.
- 110 The instruction was a *Read* for which the thread obtained the referenced data from a cache on a different chip from the thread.
- 111 The instruction was a *Store* for which the data were placed into a location other than the primary data cache.

63 **Sampled Instruction Completed** (SICMPL)

Set to 1 if the sampled instruction has completed; otherwise set to 0.

11.4.11 Other Performance Monitor Registers

SIER2, SIER3 and MMCR3 are 64-bit registers. The contents of these registers are implementation-dependent.

Programming Note

Applications are expected to access the contents of SIER2, SIER3 and MMCR3 by means of a privileged service program which will be typically invoked by a system call and which is capable of interpreting the contents of these registers, and returns to the application a generalized summary of the register contents in a form that is not implementation-dependent.

The recommended secure configuration of the Performance Monitor Facility is to disable access to group B Performance Monitor registers in problem state via setting $MMCR0_{PMCC}$ and $MMCR0_{PMCCEXT}$ appropriately, which is congruous to the programming model described above.

11.5 Branch History Rolling Buffer

The Branch History Rolling Buffer (BHRB) is described in Chapter 7 of Book II but only at the level required by application programmers. Additional aspects of the BHRB are described here.

In order to enable problem state programs to use the BHRB, $MMCR0_{BHRBA}$ must be set to 1 to enable execution of ***clrbhrb*** and ***mfbhrbe*** instructions in problem state. Additionally, $MMCR0_{PMCC}$ must be set to 0b10 or 0b11 to allow problem state programs to read and write the necessary Performance Monitor registers. (See Section 11.4.4.)

If Performance Monitor event-based branching is desired, $MMCR0_{EBE}$ 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 $MMCR0_{PMAO}$ in order to determine when a Performance Monitor alert occurs.

The BHRB is written by the hardware only in problem state (MSR_{PR}=1), and if and only if [] $MMCR0_{PMAE}$=1 and $MMCRA_{BHRBRD}$=0. After $MMCR0_{PMAE}$ has been set to 1 and a Performance Monitor alert occurs, $MMCR0_{PMAE}$ is set to 0 and the BHRB is not altered by hardware until software sets $MMCR0_{PMAE}$ to 1 again.

When $MMCR0_{PMAE}$=1, ***mfbhrbe*** instructions return 0s to the target register.

Programming Note

mfbhrbe instructions return 0s when $MMCR0_{PMAE}$=1 in order to prevent software from reading the BHRB while it is being written by hardware.

11.5.1 BHRB Filtering

When the BHRB is written by hardware, only those *Branch* instructions that meet the filtering criterion specified in $MMCRA_{IFM}$ and for which the branch was taken are included.

Filtering restricts the type of Branch instructions that are entered into the BHRB. The filtering criteria are defined using the following terminology.

- **Call**: A *Branch* instruction with the LK field set to 1.
- **Return**: A ***bclr*** instruction with the BH field set to 0s.
- **Jump**: Any *Branch* instruction that is not a call or a return.
- **Conditional Branch**: Any *Branch* instruction other than an I-Form *Branch* instruction, or a B- or XL-Form *Branch* instruction with the BO field set to "branch always." (See Figure 2.5 in Book I.)
- **Unconditional Branch**: Any *Branch* instruction other than a conditional branch instruction
- **Indirect Branch**: Any XL-Form *Branch* instruction
- **Direct Branch**: Any B- or I-Form *Branch* instruction Software is able to prevent various combinations of each of the above types of *Branch* instructions from being entered into the BHRB using the IFM field in MMCRA. (See Section 11.4.7, "Monitor Mode Control Register A".)

Software is able to prevent various combinations of each of the above types of Branch instructions from being entered into the BHRB using the IFM field in MMCRA. (See Section 11.4.7, "Monitor Mode Control Register A".)

Chapter 12. Processor Control

12.1 Overview

The Processor Control facility provides a mechanism for the ultravisor or hypervisor to send messages to other threads in the system. Privileged non-hypervisor programs are able to send messages to other threads on the same multi-threaded processor; however if the processor is configured into sub-processors, privileged non-hypervisor programs can only send messages to other threads on the same sub-processor.

12.2 Programming Model

Ultravisor-level, hypervisor-level, and privileged-level messages can be sent. Ultravisor-level messages are sent using the ***msgsndu*** instruction and cause ultravisor-level exceptions when received. Hypervisor-level messages are sent using the ***msgsnd*** instruction and cause hypervisor-level exceptions when received. Privileged-level messages are sent using the ***msgsndp*** instruction and cause privileged-level exceptions when received. For all three instructions, the message type and destination threads are specified in a General Purpose Register.

If a message is received by a thread, the exception corresponding to the message type is generated. When the exception is generated, the corresponding interrupt occurs when no higher priority exception exists and the interrupt is enabled (MSR_{EE}=1 for the Directed Privileged Doorbell interrupt, MSR_{EE}=1 or MSR_{HV}=0 for the Directed Hypervisor Doorbell interrupt, and MSR_{EE}=1 or $MSR_{S\ HV\ PR} \neq 0b110$ for the Directed Ultravisor Doorbell interrupt).

A Directed Privileged Doorbell exception remains until the corresponding interrupt occurs, or the exception is cleared by execution of a ***mtspr***(DPDES) or ***msgclrp*** instruction.

A Directed Hypervisor Doorbell exception remains until the corresponding interrupt occurs, or the exception is cleared by execution of a ***msgclr*** instruction.

A Directed Ultravisor Doorbell exception remains until the corresponding interrupt occurs, or the exception is cleared by execution of a ***msgclru*** instruction.

If a Doorbell exception of a given privilege is present and the corresponding interrupt is pended because MSR_{EE}=0, additional Doorbell exceptions of that privilege are ignored until the exception is cleared.

12.3 Processor Control Registers

12.3.1 Directed Privileged Doorbell Exception State

The layout of the Directed Privileged Doorbell Exception State (DPDES) register is shown in Figure 12.1.

Figure 12.1: Directed Privileged Doorbell Exception State Register

The DPDES register is a 64-bit register. For t $<$ T, where T is the number of threads on the sub-processor (or on the multi-threaded processor if sub-processors are not supported), bit 63-t corresponds to the thread with privileged thread number t.

The value of bit t indicates the presence of a Directed Privileged Doorbell exception on the thread with privileged thread number t. Bit t is cleared when a Directed Privileged Doorbell interrupt occurs on thread t.

When the contents of $DPDES_{63-t}$ change from 0 to 1, a Directed Privileged Doorbell exception will come into existence on privileged thread number t within a reasonable period of time. When the contents of $DPDES_{63-t}$ change from 1 to 0, the existing Directed Privileged Doorbell exception, if any, on privileged thread number t, will cease to exist within a reasonable period of time, but not later than the completion of the next context synchronizing instruction or event on privileged thread number t.

The preceding paragraph applies regardless of whether the change in the contents of $DPDES_{63-t}$ is the result a ***msgsndp*** or ***msgclrp*** instruction or of modification of the DPDES register caused by execution of an ***mtspr*** (DPDES) instruction.

Bits 0:63-T of the DPDES are reserved.

Programming Note

The primary use of the DPDES is to provide the means for the hypervisor to save a [sub-]processor's Directed Privileged Doorbell exception state when the set of programs running on the [sub-]processor is swapped out or moved from one [sub-]processor to another. Since there is no such need for a similar function for the hypervisor or ultravisor, there is no similar register for the hypervisor or ultravisor. Privileged programs are able to read the DPDES in order to poll for Directed Privileged Doorbell exceptions when the corresponding interrupt is disabled (MSR_{EE}=1).

12.4 Processor Control Instructions

msgsndu, ***msgsnd***, ***msgsndp***, ***msgclru***, ***msgclr***, and ***msgclrp*** instructions are provided for sending and clearing messages. ***msgsync*** is provided to enable the thread that is target of a ***msgsndu*** or ***msgsnd*** instruction to ensure that stores performed by the message-sending thread before it executed ***msgsndu*** or ***msgsnd*** have been performed with respect to the target thread. ***msgsndp*** and ***msgclrp*** are privileged instructions; ***msgsnd***, ***msgclr***, and ***msgsync*** are hypervisor privileged instructions; ***msgsndu*** and ***msgclru*** are ultravisor privileged instructions.

Message Send Ultravisor X-form

msgsndu RB

31	///	///	RB	78	/
0	6	11	16	21	31

```
msgtype ← GPR(RB)32:36
payload ← GPR(RB)37:63
if (msgtype = 0x05)
then
   send_msg(msgtype, payload)
```

msgsndu sends a message to other threads in the system. The message type and destination thread(s) are specified in RB.

RB

← Message Payload →

///	TYPE	B	///	PROCIDTAG
0	32	37	39	44 63

The contents of RB are defined below. Bits 37:63 are referred to as the message payload.

Field	Description
0:31	Reserved
32:36	**Type** If Type=0x05, then a Directed Ultravisor Doorbell message is to be sent to the thread(s) specified in the Message Payload field. All other values of the Type field are reserved; if the instruction is executed with this field set to a reserved value, the instruction is treated as a no-op.
37:38	**Broadcast** (B) 00 The message is sent to the thread for which $PIR_{44:63}$ is equal to the value of the PROCIDTAG field in the message payload. 01 The message is sent to all threads on the same sub-processor as the thread for which $PIR_{44:63}$ is equal to the value of the PROCIDTAG field in the message payload. 10 The message is sent to all threads on the same multi-threaded processor as the thread for which $PIR_{44:63}$ is equal to the value of the PROCIDTAG field in the message payload. 11 Reserved
39:43	Reserved
44:63	**PROCIDTAG** This field indicates the recipient thread(s) as specified in the B field. If this field set to a value that is not the same as bits $PIR_{44:63}$ 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 12.2.

This instruction is ultravisor privileged.

Special Registers Altered:

None

Programming Note

If ***msgsndu*** is used to notify the receiver that updates have been made to storage, a ***sync*** should be placed between the stores and the ***msgsndu***. See Section 6.9.2.

Message Clear Ultravisor X-form

msgclru RB

31	///	///	RB	110	/
0	6	11	16	21	31

```
t ← hypervisor thread number of executing thread
if (msgtype = 0x05) then
        clear any Directed Ultravisor Doorbell exception
for thread t
```

msgclru clears a message previously accepted by the thread executing the ***msgclru***.

Let msgtype be $(RB)_{32:36}$, and let t be the hypervisor thread number of the thread executing the ***msgclru*** instruction.

If msgtype = 0x05, then clear any Directed Ultravisor Doorbell exception that exists on thread t; otherwise, this instruction is treated as a no-op.

This instruction is ultravisor privileged.

Special Registers Altered:

None

Programming Note

msgclru is typically issued only when MSR_{EE}=0. If ***msgclru*** is executed when MSR_{EE}=1 when a Directed Ultravisor Doorbell interrupt is about to occur, the corresponding interrupt may or may not occur.

Message Send X-form

msgsnd RB

31	///	///	RB	206	/
0	6	11	16	21	31

```
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

← Message Payload →

///	TYPE	B	///	PROCIDTAG
0	32	37	39	44 63

The contents of RB are defined below. Bits 37:63 are referred to as the message payload.

Field	Description
0:31	Reserved
32:36	**Type** If Type=0x05, then a Directed Hypervisor Doorbell message is to be sent to the thread(s) specified in the Message Payload field. All other values of the Type field are reserved; if the instruction is executed with this field set to a reserved value, the instruction is treated as a no-op.
37:38	**Broadcast** (B) 00 The message is sent to the thread for which $PIR_{44:63}$ is equal to the value of the PROCIDTAG field in the message payload. 01 The message is sent to all threads on the same sub-processor as the thread for which $PIR_{44:63}$ is equal to the value of the PROCIDTAG field in the message payload. 10 The message is sent to all threads on the same multi-threaded processor as the thread for which $PIR_{44:63}$ is equal to the value of the PROCIDTAG field in the message payload. 11 Reserved
39:43	Reserved
44:63	**PROCIDTAG** This field indicates the recipient thread(s) as specified in the B field. If this field set to a value that is not the same as bits $PIR_{44:63}$ 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 12.2.

This instruction is hypervisor privileged.

Special Registers Altered:

None

Programming Note

If ***msgsnd*** is used to notify the receiver that updates have been made to storage, a ***sync*** should be placed between the stores and the ***msgsnd***. See Section 6.9.2.

Message Clear X-form

msgclr RB

31	///	///	RB	238	/
0	6	11	16	21	31

```
t ← 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 X-form

msgsndp RB

31	///	///	RB	142	/
0	6	11	16	21	31

```
msgtype ← (RB)32:36
payload ← (RB)37:63
t ← (RB)57:63
if msgtype = 5 and
   t ≤ maximum privileged thread number on
         processor or sub-processor
then
   DPDES63-t ← 1
   send_msg(msgtype, payload, t)
```

msgsndp sends a message to other threads that are on the same multi-threaded processor (if the processor is not in sub-processor mode) or to other threads that are on the same sub-processor (if the processor is in sub-processor mode). The message type and destination thread(s) are specified in RB.

RB

Message Payload

///	TYPE	///	TIRTAG
0	32	37 39	57 63

The contents of RB are defined below. Bits 37:63 are referred to as the message payload.

Bits	Description
37:56	Reserved
57:63	**TIRTAG** This message is sent to the thread for which the privileged thread number is equal to contents of the TIRTAG field of the message payload, and one of the following conditions applies. • for processors that are not partitioned into sub-processors, the message 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 message 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 12.2.

This instruction is privileged.

Special Registers Altered:

Register	*Field(s)*
DPDES	

Programming Note

If ***msgsndp*** is used to notify the receiver that updates have been made to storage, a ***lwsync*** or ***sync*** should be placed between the stores and the ***msgsndp***. See Section 6.9.2.

Message Clear Privileged X-form

msgclrp RB

31	///	///	RB	174	/
0	6	11	16	21	31

```
msgtype ← (RB)32:36
t ← privileged thread number of executing thread
IF (msgtype = 0x05)
then
   DPDES63-t ← 0
```

msgclrp clears a message previously accepted by the thread executing the ***msgclrp***.

Let msgtype be $(RB)_{32:36}$, and let t be the privileged thread number of the thread executing the ***msgclrp***.

If msgtype = 0x05, then clear any Directed Privileged Doorbell exception that exists on thread t by setting $DPDES_{63-t}$ to 0; otherwise, this instruction is treated as a no-op.

This instruction is privileged.

Special Registers Altered:

Register	*Field(s)*
DPDES	

Programming Note

msgclrp is typically issued only when MSR_{EE}=0. If ***msgclrp*** is executed when MSR_{EE}=1 when a Directed Privileged Doorbell interrupt is about to occur, the corresponding interrupt may or may not occur.

Message Synchronize X-form

msgsync

31	///	///	///	886	/
0	6	11	16	21	31

In conjunction with the *Synchronize* and ***msgsndu*** or ***msgsnd*** instructions, the ***msgsync*** instruction provides an ordering function for stores that have been performed with respect to the thread executing the *Synchronize* and ***msgsndu*** or ***msgsnd*** instructions, relative to data accesses by other threads that are performed after a Directed Ultravisor Doorbell or Directed Hypervisor Doorbell interrupt has occurred, as described in the *Synchronize* instruction description on p. 1156.

This instruction is hypervisor privileged.

Special Registers Altered:

None

Programming Note

When used in conjunction with ***msgsndu*** or ***msgsnd***, *Synchronize* with L = 0 or 2 is executed on the thread that will execute the ***msgsndu*** or ***msgsnd***, and ***msgsync*** is executed on another thread – typically the thread that is the target of the ***msgsndu*** or ***msgsnd***, but possibly any other thread (partly because the software that services the Directed Ultravisor Doorbell or Directed Hypervisor Doorbell interrupt may ultimately run on a thread other than that which received the exception). The *Synchronize* precedes the ***msgsndu*** or ***msgsnd***; the ***msgsync*** is executed after the Directed Ultravisor Doorbell or Directed Hypervisor Doorbell interrupt occurs, and precedes all instructions that need to “see” the values stored by the stores that are in set A of the memory barrier created by the *Synchronize*; see Section 6.9.2, “Synchronize Instruction ”.

Chapter 13. 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 MSR_{IR} 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 13.1 (for data access) and Table 13.2 (for instruction fetch and execution).

The notation “CSI” in the tables means any context synchronizing instruction (e.g., ***sc***, ***isync***, or ***rfid***). A context synchronizing interrupt (i.e., any interrupt except non-recoverable System Reset or non-recoverable Machine Check) can be used instead of a context synchronizing instruction. If it is, phrases like “the synchronizing instruction”, below, should be interpreted as meaning the instruction at which the interrupt occurs. If no software synchronization is required before (after) a context-altering instruction, “the synchronizing instruction before (after) the context-altering instruction” should be interpreted as meaning the context-altering instruction itself.

The synchronizing instruction before the context-altering instruction ensures that all instructions up to and including that synchronizing instruction are fetched and executed in the context that existed before the alteration. The synchronizing instruction after the context-altering instruction ensures that all instructions after that synchronizing instruction are fetched and executed in the context established by the alteration. Instructions after the first synchronizing instruction, up to and including the second synchronizing instruction, may be fetched or executed in either context.

If a sequence of instructions contains context-altering instructions and contains no instructions that are affected by any of the context alterations, no software synchronization is required within the sequence.

Programming Note

Sometimes advantage can be taken of the fact that certain events, such as interrupts, and certain instructions that occur naturally in the program, such as the ***rfid*** that returns from an interrupt handler, provide the required synchronization.

Because the instructions between the first synchronizing instruction (exclusive) and the second synchronizing instruction (inclusive) may be fetched or executed in either context, if the context alteration affects whether the second synchronizing instruction can be fetched or executed then the context alteration will not necessarily be synchronized in the manner the programmer expected. For example, if the second synchronizing instruction is in a different virtual page from the context-altering instruction, and fetching instructions from this virtual page is prohibited by Virtual Page Class Key Storage Protection, and the context-altering instruction is an ***mtiamr*** that enables fetching instructions from this virtual page, it is indeterminate whether the second synchronizing instruction will be executed or a [Hypervisor] Instruction Storage interrupt will occur instead.

No software synchronization is required before or after a context-altering instruction that is also context synchronizing or when altering the MSR in most cases (see the tables). No software synchronization is required before most of the other alterations shown in Table 13.2, because all instructions preceding the context-altering instruction are fetched and decoded before the context-altering instruction is executed (the hardware must determine whether any of these preceding instructions are context synchronizing).

Unless otherwise stated, the material in this chapter assumes a single-threaded environment.

Instruction or Event	Required Before	Required After	Notes
interrupt	none	none	
rfid	none	none	
hrfid	none	none	
urfid	none	none	
rfscv	none	none	
sc	none	none	
scv	none	none	
Trap	none	none	
mtspr (AMR)	CSI	CSI	13
mtspr (PIDR)	CSI	CSI	6,20
mtspr (DAWRn)	CSI	CSI	
mtspr (DAWRXn)	CSI	CSI	
mtspr (HRMOR)	CSI	CSI	11,17
mtspr (URMOR)	CSI	CSI	11,17
mtspr (LPCR)	CSI	CSI	14,19
mtspr (PTCR)	***ptesync***	CSI	3
mtspr (SMFCTRL)	CSI	CSI	
mtmsrd (SF)	none	none	
mtmsr[***d***] (PR)	none	none	
mtmsr[***d***] (DR)	none	none	
mtspr (LPIDR)	CSI	CSI	6,14,20
slbie	CSI	CSI	4,6
slbieg when UPRT=1	CSI	CSI	4,6
slbia	CSI	CSI	4,6
slbmte	CSI	CSI	4,10
tlbie	CSI	CSI	4,6
tlbiel	CSI	CSI	4,6
Store(PTE)	none	{***ptesync***, CSI}	5,6
Store(STE)	none	{***ptesync***, CSI}	5,6
Store(PRTE)	none	{***ptesync***, CSI}	5,6
Store(PATE)	none	{***ptesync***, CSI}	5,6,19

Table 13.1: Synchronization requirements for data access

Instruction or Event	Required Before	Required After	Notes
interrupt	none	none	
rfid	none	none	
hrfid	none	none	
urfid	none	none	
rfscv	none	none	
sc	none	none	
scv	none	none	
Trap	none	none	
mtmsrd (SF)	none	none	7
mtmsr[***d***] (EE)	none	none	1
mtmsr[***d***] (PR)	none	none	8
mtmsr[***d***] (FP)	none	none	
mtmsr[***d***](FE0,FE1)	none	none	
mtmsr[***d***] (TE)	none	none	
mtmsr[***d***] (IR)	none	none	8
mtmsr[***d***] (RI)	none	none	
mtspr (DEC)	none	none	9
mtspr (PIDR)	CSI	CSI	6,20
mtspr (IAMR)	none	CSI	
mtspr (CTRL)	none	none	
mtspr (FSCR)	none	CSI	
mtspr (DPDES)	none	CSI	17
mtspr (CIABR)	none	CSI	
mtspr (HFSCR)	none	CSI	
mtspr (HDEC)	none	none	9
mtspr (HRMOR)	none	CSI	8,11,17
mtspr (URMOR)	none	CSI	8,11,17
mtspr (LPCR)	none	CSI	12,14,19
mtspr (LPIDR)	CSI	CSI	6, 14, 17, 20
mtspr (PCR)	none	CSI	17
mtspr (PTCR)	***ptesync***	CSI	3,17
mtspr (SMFCTRL)	none	CSI	
mtspr (Perf. Mon.)	none	CSI	15,18
mtspr (BESCR)	none	CSI	16,18
mtspr(HDEXCR[ENF])	none	CSI	21
mtspr(DEXCR[PRO])	none	CSI	21
mtspr(HDEXCR[HNU])	none	{***isync***, ***exser***}	21
mtspr(DEXCR[PNH])	none	{***isync***, ***exser***}	22
mtspr (HASHKEYR)	none	CSI	
mtspr (HASHPKEYR)	none	CSI	
slbie	none	CSI	4,6
slbieg when UPRT=1	none	CSI	4,6
slbia	none	CSI	4,6
slbmte	none	CSI	4,8,10
tlbie	none	CSI	4,6
tlbiel	none	CSI	4,6
Store(PTE)	none	{***ptesync***, CSI}	5,6,8
Store(STE)	none	{***ptesync***, CSI}	5,6,8
Store(PRTE)	none	{***ptesync***, CSI}	5,6,8
Store(PATE)	none	{***ptesync***, CSI}	5,6,8,19

Table 13.2: Synchronization requirements for instruction fetch and/or execution

Notes:

1. The effect of changing the EE bit is immediate, even if the ***mtmsr***[***d***] instruction is not context synchronizing (i.e., even if L=1).
   - If an ***mtmsr***[***d***] instruction sets the EE bit to 0, neither an External interrupt, a Decrementer interrupt nor a Performance Monitor interrupt occurs after the ***mtmsr***[***d***] is executed.
   - If an ***mtmsr***[***d***] instruction changes the EE bit from 0 to 1 when an External, Decrementer, Performance Monitor or higher priority exception exists, the corresponding interrupt occurs immediately after the ***mtmsr***[***d***] is executed, and before the next instruction is executed in the program that set EE to 1.
   - If a hypervisor executes the ***mtmsr***[***d***] instruction that sets the EE bit to 0, a Hypervisor Decrementer interrupt does not occur after ***mtmsr***[***d***] is executed as long as the thread remains in hypervisor state.
   - If the hypervisor executes an ***mtmsr***[***d***] instruction that changes the EE bit from 0 to 1 when a Hypervisor Decrementer or higher priority exception exists, the corresponding interrupt occurs immediately after the ***mtmsr***[***d***] instruction is executed, and before the next instruction is executed, provided HDICE is 1.
2. Synchronization requirements for this instruction are implementation-dependent.
3. The PTCR controls all implicit and explicit storage accesses performed by all threads on the processor when the thread is not in hypervisor or ultravisor real addressing mode. Modifying the PTCR requires that the following conditions be achieved on all threads on the processor.
   - the thread is in hypervisor or ultravisor real addressing mode
   - all previous accesses (implicit and explicit) initiated when the thread was not in hypervisor or ultravisor real addressing mode have been performed with respect to all threads
   - no subsequent accesses which require translation have been initiated
4. For data accesses, the context synchronizing instruction before the ***slbie***, ***slbieg***, ***slbia***, ***slbmte***, ***tlbie***, or ***tlbiel*** instruction ensures that all preceding instructions that access data storage have completed to a point at which they have reported all exceptions they will cause.

   The context synchronizing instruction after the ***slbie***, ***slbieg***, ***slbia***, ***tlbie*** or ***tlbiel*** instruction ensures that storage accesses associated with instructions following the context synchronizing instruction will not use the SLB entry(s), TLB entry(s), or implementation-specific lookaside information being invalidated.
5. The notation "{***ptesync***,CSI}" denotes an instruction sequence. Other instructions may be interleaved with this sequence, but these instructions must appear in the order shown.

   No software synchronization is required before the *Store* instruction because (a) stores are not performed out-of-order and (b) address translations associated with instructions preceding the *Store* instruction are not performed again after the store has been performed (see Sections 6.5 and 6.10). These properties ensure that all address translations associated with instructions preceding the *Store* instruction will be performed using the old contents of the PTE.

   The ***ptesync*** instruction after the *Store* instruction ensures that all searches of the Page Table that are performed after the ***ptesync*** instruction completes will use the value stored (or a value stored subsequently). The context synchronizing instruction after the ***ptesync*** instruction ensures that any address translations associated with instructions following the context synchronizing instruction that were performed using the old contents of the PTE will be discarded, with the result that these address translations will be performed again and, if there is no corresponding entry in any TLB, SLB, page walk cache, cache of Partition or Process Table entries, or implementation-specific address translation lookaside information, will use the value stored (or a value stored subsequently).

   []
6. Additional software synchronization may be required for the use of this instruction in multi-threaded environments and/or when storage management operations such as gathering Reference and Change bit values or changing EA to RA mappings are being performed. For example, it may be necessary to invalidate one or more SLB entries and/or TLB entries on all threads in the system and to be able to determine that the invalidations have completed and that all accesses that used or were associated with the invalidated translations have been performed. This requires using ***slbsync*** or ***tlbsync*** in multi-threaded environments, and using ***ptesync*** in all environments.

   Section 6.10 describes the use of ***slbie***, ***slbieg***, ***slbia***, ***tlbiel***, ***tlbie***, *Store*, and related instructions to maintain the Segment Table and Page Table, in both multi-threaded environments and environments consisting of only a single-threaded processor.

Programming Note

In a multi-threaded system, if software locking is used to help ensure that the requirements described in Section 6.10 are satisfied, the ***isync*** instruction near the end of the lock acquisition sequence (see Section B.2.1.1 of Book II) may naturally provide the context synchronization that is required before the alteration.

7. The alteration must not cause an implicit branch in effective address space. Thus, when changing MSR_{SF} from 1 to 0, the ***mtmsrd*** instruction must have an effective address that is less than 2^{32} - 4. Furthermore, when changing MSR_{SF} from 0 to 1, the ***mtmsrd*** instruction must not be at effective address 2^{32} - 4 (see Section 6.3.2 on page 1111).
8. The alteration must not cause an implicit branch in real address space. Thus the real address of the context-altering instruction and of each subsequent instruction, up to and including the next context synchronizing instruction, must be independent of whether the alteration has taken effect.

Programming Note

If it is desired to set MSR_{IR} to 1 early in an operating system interrupt handler, advantage can sometimes be taken of the fact that $EA_{0:3}$ are ignored when forming the real address when address translation is disabled and $MSR_{HV} = 0$. For example, if address translation resources are set such that effective address 0xn000_0000_0000_0000 maps to real address 0x000_0000_0000_0000 when address translation is enabled, where n is an arbitrary 4-bit value, the following code sequence, in real page 0, can be used early in the interrupt handler.

```
      la      rx,target
      li      ry,0xn000
      sldi    ry,ry,48
      or      rx,rx,ry # set high-order nibble
                       # of target addr to 0xn
      mtctr   rx
      bcctr            # branch to targ

targ: mfmsr   rx
      ori     rx,rx,0x0020
      mtmsrd  rx        # set MSR_IR to 1
```

The ***mtmsrd*** does not cause an implicit branch in real address space because the real address of the next sequential instruction is independent of MSR_{IR}. Using ***mtmsrd***, rather than ***rfid*** (or similar context synchronizing instruction that alters the control flow), may yield better performance on some implementations.

(Variations on the technique are possible. For example, the target instruction of the ***bcctr*** can be in arbitrary real page P, where P is a 48-bit value, provided that effective address 0xn || P || 0x000 maps to real address P || 0x000 when address translation is enabled.)

9. The elapsed time between the contents of the Decrementer or Hypervisor Decrementer becoming negative and the signaling of the corresponding exception is not defined.
10. If an ***slbmte*** instruction alters the mapping, or associated attributes, of a currently mapped ESID, the ***slbmte*** must be preceded by an ***slbie*** (or ***slbia***) instruction that invalidates the existing translation. This applies even if the corresponding entry is no longer in the SLB (the translation may still be in implementation-specific address translation lookaside information). No software synchronization is needed between the ***slbie*** and the ***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 ***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 URMOR or the HRMOR is modified, software must invalidate all implementation-specific lookaside information used in address translation that depends on the old contents of the register (i.e., the contents immediately before the modification). The ***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 $LPCR_{MER}$ = 1 does not necessarily ensure that the exception effects of $LPCR_{MER}$ are consistent with the contents of $LPCR_{MER}$. 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 $LPCR_{HR}$ must not be altered when MSR_{DR}=1 or MSR_{IR}=1; if they are, the results are undefined.

Programming Note

For instruction fetch, the prohibitions above are because of the difficulty of avoiding an implicit branch relative to the value of enabling software to avoid using hypervisor real addressing mode for the operation. For data access, the prohibitions above are to avoid errant (wrongly timed and/or for an incorrect context) speculative translation in support of hardware data prefetching. (The tables used for translation are determined by the partition ID and $LPCR_{HR}$ is used as a shortcut. See Section 6.7.6 for details.)

15. This line applies to the following Performance Monitor SPRs: PMC1-6, MMCR0, MMCR1, MMCR2, and MMCRA.
16. This line applies to all SPR numbers that access the BESCR (800-803, 806).
17. There are additional software synchronization requirements when an ***mtspr*** instruction modifies this SPR in a multi-threaded environment. See Section 2.7.
18. As an alternative to a CSI, the execution of an ***rfebb*** instruction or the occurrence of an event-based branch is sufficient to provide the necessary synchronization.
19. When $LPCR_{ISL}$ or $PATE_{PS}$ is modified, software must invalidate all implementation-specific lookaside information used in address translation that depends on the old contents of the field (i.e., the contents immediately before the modification). The ***slbia*** instruction can be used to invalidate all such implementation-specific lookaside information.
20. If the current contents of LPIDR correspond to a partition that uses Radix Tree translation, there must be an ***hwsync*** between the last instruction for which the address of an associated storage access was translated using the current contents of LPIDR and a subsequent ***mtlpidr***, and similarly for PIDR and ***mtpidr***. For the similar situation when the current contents of LPIDR correspond to a partition that uses HPT translation, no additional synchronization is required preceding ***mtpidr***, but ***ptesync*** must be used instead of ***hwsync*** prior to ***mtlpidr***, and the thread must be in hypervisor state with address translation disabled when the ***ptesync*** is executed and must remain in that condition until after the ***mtlpidr*** is executed.

Programming Note

The preceding requirement permits designs to optimize ***tlbie*** processing when the LPID and/or PID values specified by the ***tlbie*** differ from those in the corresponding register on the receiving thread, by ensuring, in software, that all storage accesses, by the receiving thread, for which the address was translated using the current contents of LPIDR and/or PIDR, and all Reference and Change bit updates associated with address translations, by the receiving thread, that used the current contents of LPIDR and/or PIDR, have been performed with respect to all threads before the contents of these registers are changed.

The more stringent requirements for partitions that use HPT translation are needed in order to prevent a Reference or Change bit update, that was performed out-of-order, from corrupting a new PTE that was created, by the thread executing the ***tlbie***, at the same real address as the PTE that was used for the address translation (performed out-of-order) with which the Reference or Change bit update was associated, or from being performed entirely in the new PTE. (***hwsync*** does not order these updates; ***ptesync*** does.) The more stringent requirements are not needed for partitions that use Radix Tree translation because for Radix Tree translation Reference and Change bit updates are performed atomically. (If an attempt by hardware to update a Reference or Change bit atomically and out-of-order fails, an [H]DSI or [H]ISI will not occur, because such interrupts do not occur out-of-order.)

If changing the contents of LPIDR and/or PIDR is part of dispatching a new program on the thread, the ***hwsync*** or ***ptesync*** just described can, if placed properly, also serve as the [***p***]***hwsync*** or ***ptesync*** that is required when a new program is dispatched, as described in the first Programming Note in Section 7.4.3. If advantage is taken of this commonality, the stronger of the two required sync variants must be used. E.g., if the current note requires ***hwsync*** and the Programming Note requires ***ptesync***, ***ptesync*** must be used.

Because for Radix Tree translation addresses can be translated using LPID (PID) value 0 without that value being in LPIDR (PIDR), the optimization described above is not possible for a ***tlbie*** instruction for which effR=1 and either (a) the specified LPID value is 0, or (b) the specified LPID value is equal to the contents of LPIDR and the specified PID value is 0. As a consequence, no ***hwsync*** is needed before changing the contents of LPIDR or PIDR from 0 to a non-zero value when Radix Tree translation is in use.

21. Changes to values of ENF bits in the HDEXCR and PRO bits in the DEXCR can only affect instruction fetch / execution behavior in a lower privilege state than the privilege state in which the ***mtspr*** is executed. The requisite *Return from* instruction is sufficient synchronization for such changes.
22. The notation "{***isync***, ***exser***}" denotes an instruction sequence. Other instructions may be interleaved within this sequence, but the specified instructions must appear in the order shown. In contrast to the changes described in the note immediately above, changes to values of HNU and PNH may take effect immediately, making additional software synchronization useful in those cases.

Power ISA Book I-III Appendices

Appendix A. Notes on the Removal of Transactional Memory from the Architecture

Facilities that are available in problem state are generally not removed from the architecture because of the potential impact to the code base. Transactional Memory is a special case in that the major operating environments require software to check the availability of the facility prior to using it. In addition, the requirement that every transaction have a failure handler means that any application that uses TM will continue to function on a degenerate TM implementation that simply fails each transaction on the instruction that follows ***tbegin.***. These two facts enable a minimally-disruptive ecosystem transition to remove TM, even while continuing to provide product features such as live partition migration.

The anticipated system support for the removal of TM uses a *synthetic TM* implementation when the thread is running in "Version 3.0 mode" ($PCR_{v3.0\ v2.07}$ = 0b10) or in "Version 2.07 mode" ($PCR_{v3.0\ v2.07}$ = 0b11). (See Section 2.5 of Book III.) For partitions running in "Version 3.1 mode" ($PCR_{v3.0\ v2.07}$ = 0b00), hypervisor software disables TM by setting $HFSCR_{58}$ (formerly the Transactional Memory Facility bit) to zero, and then ensures that the thread will always be in Non-transactional state by setting $MSR_{29:30}$ (formerly the Transaction State field) to 0b00 before dispatching the partition. (The latter requirement will be satisfied naturally provided that the hypervisor executes ***treclaim.*** as part of context switch, as required in Versions 2.07B and 3.0C of the architecture.)

Synthetic TM is required in order to provide unrestricted live migration of partitions running in PCR modes that supported TM ($PCR_{v3.0}$=1). The TM architecture allowed for access to TM SPRs and execution of TM instructions even in Non-transactional state, making pure, interrupt-driven emulation too great a performance risk. Instead, processor implementations provide default behaviors for TM-related operations. Each new transaction fails on the instruction after ***tbegin.***, going immediately to the failure handler. (Failure recording for this special case sets the Failure Persistent bit in TEXASR to reduce the likelihood that software will pointlessly retry the transaction, and also sets the Implementation-specific cause bit in TEXASR. The ***tbegin.*** is not traced, regardless of the contents of MSR_{TE} and CIABR.) ***mtspr*** and ***mfspr*** for TM SPRs have their normal behavior. Most of the other TM instructions behave as they would have behaved in Non-transactional state – i.e., just changing CR0 and FXCC. (See Book II Section 5.5 and Book III Section 5.4.4 in Version 3.0C for background on the TM instructions.) These are the behaviors that will be seen by most software. Thus, under synthetic TM, a thread is never productively in Transactional state, and is in Suspended state (actually, synthetic Suspended state) only if the program was migrated, while in Suspended state, from a processor that implements Version 3.0C or Version 2.07B. The handling for applications migrated while in Suspended state is beyond the scope of this explanation. See the implementation's user manual for details.

The architecture proper portrays TM to be fully removed. The HFSCR is used to disable TM in Version 3.1 mode because it is impractical to give the appearance that TM has been entirely removed. When the thread is in problem state or privileged non-hypervisor state, the deviations from the behavior for complete removal will be what one would expect if TM was implemented but disabled by the HFSCR. The following sections describe deviations in Version 3.1 mode with $HFSCR_{58}$ set to zero from the behavior if TM was completely removed. The description is somewhat redundant in describing deviations both from the cause and effect points of view. It does not discuss deviations that can happen only in hypervisor state. The hypervisor and ultravisor must not use any resource formerly allocated to TM other than $HFSCR_{58}$.

A.1 Attempted Execution of TM Instructions

Non-privileged TM instructions were encoded using primary opcode 31 with extended opcodes 654, 686, 718, 750, 782, 814, 846, 878, and 910. Privileged TM instructions were encoded using primary opcode 31 with extended opcodes 942 and 1006.

An attempt in privileged non-hypervisor state to execute a TM instruction, or an attempt in problem state to execute a non-privileged TM instruction, will cause a Hypervisor Facility Unavailable interrupt with $HFSCR_{IC}$ = 0x05 indicating that the cause was an attempt to use a TM resource. Since the architecture portrays the instructions as reserved, a Hypervisor Emulation Assistance interrupt

would be expected. The hypervisor's Hypervisor Facility Unavailable interrupt handler must pass the interrupt to the operating system as if the instruction had caused an "Illegal Instruction type Program Interrupt" just as the hypervisor's HEAI handler would.

An attempt in problem state to execute a privileged TM instruction will cause a Privileged Instruction type Program interrupt (because a Privileged Instruction type Program interrupt has higher priority than a Hypervisor Facility Unavailable interrupt). Since the architecture portrays the instructions as reserved, an HEAI would be expected. There is no way for the hypervisor to intercept this interrupt to pass it to the operating system as if the instruction had caused an "Illegal Instruction type Program interrupt". Moreover, there is no easy way for the OS's Program interrupt handler to determine that the instruction that caused the Privileged Instruction type Program interrupt is a TM instruction and then to handle that case as if the instruction were illegal. In the major operating environments the two kinds of error are reported to the application program in a manner that differs only slightly between the two, so this deviation is deemed acceptable.

A.2 Attempted Access of a TM SPR

The TM SPRs were SPRs 128, 129, and 130, with 131 as a partial alias to 130 for 32-bit software. (They were all non-privileged.)

An attempt in problem state, or in privileged non-hypervisor state when $LPCR_{EVIRT}$=1, to access a TM SPR will cause a Hypervisor Facility Unavailable interrupt with $HFSCR_{IC}$ = 0x05. Since the architecture portrays the SPRs as "undefined for the implementation" (see the instruction descriptions for ***mtspr*** and ***mfspr*** in Section 5.4.4 of Book III), an HEAI would be expected. The hypervisor's Hypervisor Facility Unavailable interrupt handler must pass the interrupt to the operating system as if the instruction had caused an "Illegal Instruction type Program interrupt" just as the hypervisor's HEAI handler would.

An attempt in privileged non-hypervisor state when $LPCR_{EVIRT}$=0 to access a TM SPR will cause a Hypervisor Facility Unavailable interrupt (with $HFSCR_{IC}$ = 0x05). Since the architecture portrays the SPR numbers as undefined, the implementation would be expected to treat the instruction as a no-op. There is no easy way for the hypervisor to determine the interrupt's cause in this case. (The interrupt does not set the HEIR.) The hypervisor's Hypervisor Facility Unavailable interrupt handler should pass the interrupt to the operating system as if the instruction had caused an "Illegal Instruction type Program interrupt" to avoid the complexity of determining its cause.

A.3 Occurrence of the Hypervisor Facility Unavailable Interrupt with HFSCR[IC]=0x05

A Hypervisor Facility Unavailable interrupt that set $HFSCR_{IC}$ to 0x05 indicated an attempt to execute a TM instruction or to access a TM SPR when $HFSCR_{58}$ is set to zero. Despite that the architecture no longer includes Transactional Memory, this variant of the Hypervisor Facility Unavailable interrrupt will occur in the circumstances described in the previous two sections and should be handled as described there.

A.4 Occurrence of the TM Bad Thing Type Program Interrupt

A Program interrupt that set $SRR1_{42}$ to 1 indicated a TM Bad Thing type Program interrupt. Despite that the architecture no longer includes Transactional Memory, this variant of Program interrupt will occur as the result of an attempt to set $MSR_{29:30}$ to a value other than 0b00 via an ***mtmsrd*** or a "return from"-type instruction, including ***rfebb***. These bits had indicated the Transaction State. As they are now reserved, the Program interrupt would be unanticipated. Because the connection between the BESCR and the MSR bits is especially hard to make without referencing previous versions of the architecture, the forms of ***rfebb*** that set $MSR_{29:30}$ to a value other than 0b00 are specified to be treated as though the instruction form is invalid. If a Program interrupt occurs from problem state with $SRR1_{42}$ = 1, the Program interrupt handler should treat the offending instruction as an illegal instruction.

A.5 Failure of Performance Monitor Counters to Count

$MMCR0_{47}$ was formerly the Freeze Counters in Non-Transactional State bit. Despite the architecture's portrayal of this bit as reserved, a 1 value in this bit will prevent Performance Monitor Counters from counting their assigned events. Software should set $MMCR0_{47}$ to zero.

A.6 Behavior of SPR Bits Formerly Related to TM

Aside from handling the bits formerly related to TM as described above, a good general rule is to practice read-modify-write on the containing SPRs for TM-related bits, leaving their values unchanged. Software is generally permitted to write reserved bits with the expectation, in privileged state, of reading back the written value if the bit is implemented in hardware and with the expectation that the bit will be ignored by hardware. Capriciously setting the relevant bits in SRR1, the CTR, or directly in the MSR may have the result discussed in Section A.4. Beyond this, the deviations take the form of hardware occasionally setting bits, such as the bits that were formerly MSR_{TM} and $BESCR_{TS}$, to zero.

Appendix B. Illegal Instructions

With the exception of the instruction consisting entirely of binary 0s, the instructions in this class are available for future extensions of the Power ISA; that is, some future version of the Power ISA may define any of these instructions to perform new functions.

The following primary opcodes are illegal.

5

The following primary opcode is used for an instruction prefix.

1

The following primary opcodes have unused extended opcodes. Their unused extended opcodes can be determined from the opcode maps in Appendix D of Book Appendices. All unused extended opcodes are illegal.

4, 6, 19, 30, 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.

primary / extended opcode
4 / 0b10110_000001
4 / 0b11110_000001
4 / 0b11000_000010
60 / 0b01011_01000.
60 / 0b10101_1011..
60 / 0b11101_1011..
63 / 0b11001_00100.
63 / 0b11010_00100.
63 / 0b10010_00111.

An instruction consisting entirely of binary 0s is illegal, and is guaranteed to be illegal in all future versions of this architecture.

Appendix C. Reserved Instructions

The instructions in this class are allocated to specific purposes that are outside the scope of the Power ISA.

The following types of instruction are included in this class.

1. The instruction having primary opcode 0, except the instruction consisting entirely of binary 0s (which is an illegal instruction; see Section 1.7.2, “Illegal Instruction Class” on page 24) and the extended opcode shown below.

   256 Service Processor “Attention”

2. Instructions from the POWER Architecture that have not been included in the Power ISA.
3. Instructions defined in a previous version of the Power ISA that have been removed.
4. Implementation-specific instructions used to conform to the Power ISA specification.
5. Any other implementation-dependent instructions that are not defined in the Power ISA.

Appendix D. Opcode Maps

This appendix contains opcode maps showing the primary opcodes, extended opcodes, and expanded opcodes.

Table D.1 describes the conventions used in the opcode maps.

Table D.1: Opcode Maps Legend

po

primary opcode (decimal format)

xop

extended or expanded opcode image (binary format)

0	instruction bit corresponding to an extended/expanded opcode bit having value of 0
1	instruction bit corresponding to an extended/expanded opcode bit having value of 1
/	reserved instruction bit, must have value of 0, otherwise invalid form
.	instruction bit corresponding to an operand or control bit, can have a value of either 0 or 1

book

Book instruction defined

version

ISA version instruction introduced

privilege

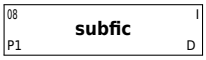
P	privileged instruction
HV	hypervisor-privileged instruction
UV	ultravisor-privileged instruction

format

instruction format

Illegal opcode

Opcode having no previous or current assignment, available for future use

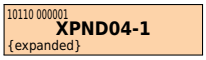


Defined opcode (primary, extended, or expanded)

Opcode assigned to a defined instruction

Primary opcode having an extended opcode field

Opcode having extended opcode field used to identify multiple instructions



Extended opcode having an expanded opcode field

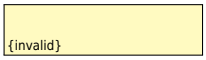
Opcode having expanded opcode field used to identify multiple instructions

Reserved opcode (primary, extended, or expanded)

Opcode is not available for future use without careful consideration

1. Opcode corresponds to an instruction defined in a previous version of the architecture that has been subsequently removed from the architecture. The opcode is treated as an illegal opcode.
2. Or, opcode is reserved for implementation-dependent use.

These opcodes will not be assigned a meaning in the Power ISA except after careful consideration of the effect of such assignment on existing implementations.



Invalid form opcode

Opcode corresponding to a defined instruction encoding with one or more reserved opcode bits having a value of 1

Table D.2: Primary Opcode Map for Opcode Space 0 (32-bit instruction encoding) (bits 0-5)

Primary opcodes of word instructions are mapped to opcode space 0

Primary opcodes of suffixes of M[M](LS|RR)-form prefixed instructions are mapped to opcode space 0

	000	001	010	011	100	101	110	111	
000	0-00 {reserved}	0-01 **PREFIX** {prefixed}	0-02 **tdi** PPC D	0-03 **twi** P1 D	0-04 **EXT004** {extended}	0-05	0-06 **EXT006** {extended}	0-07 **mulli** P1 D	000
001	0-08 **subfic** P1 D	0-09 {reserved}	0-10 **cmpli** P1 D	0-11 **cmpi** P1 D	0-12 **addic** P1 D	0-13 **addic.** P1 D	0-14 **[p]addi** P1/v3.1 [MLS:]D	0-15 **addis** P1 D	001
010	0-16 **bc[l][a]** P1 B	0-17 **EXT017** {extended}	0-18 **b[l][a]** P1 I	0-19 **EXT019** {extended}	0-20 **rlwimi[.]** P1 M	0-21 **rlwinm[.]** P1 M	0-22 **OP sandbox** {reserved/}v3.0C	0-23 **rlwnm[.]** P1 M	010
011	0-24 **ori** P1 D	0-25 **oris** P1 D	0-26 **xori** P1 D	0-27 **xoris** P1 D	0-28 **andi.** P1 D	0-29 **andis.** P1 D	0-30 **EXT030** {extended}	0-31 **EXT031** {extended}	011
100	0-32 **[p]lwz** P1/v3.1 [MLS:]D	0-33 **lwzu** P1 D	0-34 **[p]lbz** P1/v3.1 [MLS:]D	0-35 **lbzu** P1 D	0-36 **[p]stw** P1/v3.1 [MLS:]D	0-37 **stwu** P1 D	0-38 **[p]stb** P1/v3.1 [MLS:]D	0-39 **stbu** P1 D	100
101	0-40 **[p]lhz** P1/v3.1 [MLS:]D	0-41 **lhzu** P1 D	0-42 **[p]lha** P1/v3.1 [MLS:]D	0-43 **lhau** P1 D	0-44 **[p]sth** P1/v3.1 [MLS:]D	0-45 **sthu** P1 D	0-46 **lmw** P1 D	0-47 **stmw** P1 D	101
110	0-48 **[p]lfs** P1/v3.1 [MLS:]D	0-49 **lfsu** P1 D	0-50 **[p]lfd** P1/v3.1 [MLS:]D	0-51 **lfdu** P1 D	0-52 **[p]stfs** P1/v3.1 [MLS:]D	0-53 **stfsu** P1 D	0-54 **[p]stfd** P1/v3.1 [MLS:]D	0-55 **stfdu** P1 D	110
111	0-56 **lq** v2.03 DQ	0-57 **EXT057** {extended}	0-58 **EXT058** {extended}	0-59 **EXT059** {extended}	0-60 **EXT060** {extended}	0-61 **EXT061** {extended}	0-62 **EXT062** {extended}	0-63 **EXT063** {extended}	111
	000	001	010	011	100	101	110	111	

Table D.3: Primary Opcode Map for Opcode Space 1 (64-bit instruction encoding) (suffix bits 0-5)

Primary opcodes of suffixes of 8[M](LS|RR)-form prefixed instructions are mapped to opcode space 1

	000	001	010	011	100	101	110	111	
000	1-00 {reserved}	1-01 {reserved}	1-02	1-03	1-04	1-05 {reserved}	1-06	1-07	000
001	1-08	1-09 {reserved}	1-10	1-11	1-12	1-13	1-14	1-15	001
010	1-16	1-17	1-18	1-19	1-20	1-21	1-22 {reserved}	1-23	010
011	1-24	1-25	1-26	1-27	1-28	1-29	1-30	1-31	011
100	1-32 **EXT132** {extended}	1-33 **EXT133** {extended}	1-34 **EXT134** {extended}	1-35	1-36	1-37	1-38	1-39	100
101	1-40	1-41 **plwa** v3.1 8LS:D	1-42 **plxsd** v3.1 8LS:D	1-43 **plxssp** v3.1 8LS:D	1-44	1-45	1-46 **pstxsd** v3.1 8LS:D	1-47 **pstxssp** v3.1 8LS:D	101
110	1-48	1-49	1-50 **plxv** v3.1 8LS:D	1-51 **plxv** v3.1 8LS:D	1-52	1-53	1-54 **pstxv** v3.1 8LS:D	1-55 **pstxv** v3.1 8LS:D	110
111	1-56 **plq** v3.1 8LS:D	1-57 **pld** v3.1 8LS:D	1-58 **plxvp** v3.1 8LS:D	1-59	1-60 **pstq** v3.1 8LS:D	1-61 **pstd** v3.1 8LS:D	1-62 **pstxvp** v3.1 8LS:D	1-63	111
	000	001	010	011	100	101	110	111	

Table D.4: PREFIX: Opcode Map (64-bit instruction encoding) (prefix bits 6:11)

	000	001	010	011	100	101	110	111	
000	00 0... **8LS-form** v3.1								000
001									001
010	01 0000 **8RR-form** v3.1								010
011									011
100	10 0... **MLS-form** v3.1								100
101									101
110	11 0000 **MRR-form** v3.1								110
111		11 1001 **MMIRR-form** v3.1							111
	000	001	010	011	100	101	110	111	

Table D.5: EXT004: Extended Opcode Map for Opcode Space 0, Primary Opcode 4 (bits 21:31)

	000000	000001	000010	000011	000100	000101	000110	000111	
00000	00000 000000 **vaddubm** v2.03 VX	00000 000001 **vmul10cuq** v3.0 VX	00000 000010 **vmaxub** v2.03 VX		00000 000100 **vrlb** v2.03 VX	00000 000101 **vrlq** v3.1 VX	.0000 000110 **vcmpequb[.]** v2.03 VC	.0000 000111 **vcmpneb[.]** v3.0 VC	00000
00001	00001 000000 **vadduhm** v2.03 VX	00001 000001 **vmul10ecuq** v3.0 VX	00001 000010 **vmaxuh** v2.03 VX		00001 000100 **vrlh** v2.03 VX	00001 000101 **vrlqmi** v3.1 VX	.0001 000110 **vcmpequh[.]** v2.03 VC	.0001 000111 **vcmpneh[.]** v3.0 VC	00001
00010	00010 000000 **vadduwm** v2.03 VX		00010 000010 **vmaxuw** v2.03 VX		00010 000100 **vrlw** v2.03 VX	00010 000101 **vrlwmi** v3.0 VX	.0010 000110 **vcmpequw[.]** v2.03 VC	.0010 000111 **vcmpnew[.]** v3.0 VC	00010
00011	00011 000000 **vaddudm** v2.07 VX		00011 000010 **vmaxud** v2.07 VX		00011 000100 **vrld** v2.07 VX	00011 000101 **vrldmi** v3.0 VX	.0011 000110 **vcmpeqfp[.]** v2.03 VC	.0011 000111 **vcmpequd[.]** v2.07 VC	00011
00100	00100 000000 **vadduqm** v2.07 VX	00100 000001 **vcmpuq** v3.1 VX	00100 000010 **vmaxsb** v2.03 VX		00100 000100 **vslb** v2.03 VX	00100 000101 **vslq** v3.1 VX		.0100 000111 **vcmpnezb[.]** v3.0 VC	00100
00101	00101 000000 **vaddcuq** v2.07 VX	00101 000001 **vcmpsq** v3.1 VX	00101 000010 **vmaxsh** v2.03 VX		00101 000100 **vslh** v2.03 VX	00101 000101 **vrlqnm** v3.1 VX		.0101 000111 **vcmpnezh[.]** v3.0 VC	00101
00110	00110 000000 **vaddcuw** v2.03 VX		00110 000010 **vmaxsw** v2.03 VX		00110 000100 **vslw** v2.03 VX	00110 000101 **vrlwnm** v3.0 VX		.0110 000111 **vcmpnezw[.]** v3.0 VC	00110
00111			00111 000010 **vmaxsd** v2.07 VX		00111 000100 **vsl** v2.03 VX	00111 000101 **vrldnm** v3.0 VX	.0111 000110 **vcmpgefp[.]** v2.03 VC	.0111 000111 **vcmpequq[.]** v3.1 VC	00111
01000	01000 000000 **vaddubs** v2.03 VX	01000 000001 **vmul10uq** v3.0 VX	01000 000010 **vminub** v2.03 VX		01000 000100 **vsrb** v2.03 VX	01000 000101 **vsrq** v3.1 VX	.1000 000110 **vcmpgtub[.]** v2.03 VC		01000
01001	01001 000000 **vadduhs** v2.03 VX	01001 000001 **vmul10euq** v3.0 VX	01001 000010 **vminuh** v2.03 VX		01001 000100 **vsrh** v2.03 VX		.1001 000110 **vcmpgtuh[.]** v2.03 VC		01001
01010	01010 000000 **vadduws** v2.03 VX		01010 000010 **vminuw** v2.03 VX		01010 000100 **vsrw** v2.03 VX		.1010 000110 **vcmpgtuw[.]** v2.03 VC	.1010 000111 **vcmpgtuq[.]** v3.1 VC	01010
01011			01011 000010 **vminud** v2.07 VX		01011 000100 **vsr** v2.03 VX		.1011 000110 **vcmpgtfp[.]** v2.03 VC	.1011 000111 **vcmpgtud[.]** v2.07 VC	01011
01100	01100 000000 **vaddsbs** v2.03 VX		01100 000010 **vminsb** v2.03 VX		01100 000100 **vsrab** v2.03 VX	01100 000101 **vsraq** v3.1 VX	.1100 000110 **vcmpgtsb[.]** v2.03 VC		01100
01101	01101 000000 **vaddshs** v2.03 VX	01101 000001 **bcdcpsgn.** v3.0 VX	01101 000010 **vminsh** v2.03 VX		01101 000100 **vsrah** v2.03 VX		.1101 000110 **vcmpgtsh[.]** v2.03 VC		01101
01110	01110 000000 **vaddsws** v2.03 VX		01110 000010 **vminsw** v2.03 VX		01110 000100 **vsraw** v2.03 VX		.1110 000110 **vcmpgtsw[.]** v2.03 VC	.1110 000111 **vcmpgtsq[.]** v3.1 VC	01110
01111			01111 000010 **vminsd** v2.07 VX		01111 000100 **vsrad** v2.07 VX		.1111 000110 **vcmpbfp[.]** v2.03 VC	.1111 000111 **vcmpgtsd[.]** v2.07 VC	01111
10000	10000 000000 **vsububm** v2.03 VX	1.000 000001 **bcdadd.** v2.07 VX	10000 000010 **vavgub** v2.03 VX	10000 000011 **vabsdub** v3.0 VX	10000 000100 **vand** v2.03 VX		.0000 000110 **vcmpequb[.]** v2.03 VC	.0000 000111 **vcmpneb[.]** v3.0 VC	10000
10001	10001 000000 **vsubuhm** v2.03 VX	1.001 000001 **bcdsub.** v2.07 VX	10001 000010 **vavguh** v2.03 VX	10001 000011 **vabsduh** v3.0 VX	10001 000100 **vandc** v2.03 VX		.0001 000110 **vcmpequh[.]** v2.03 VC	.0001 000111 **vcmpneh[.]** v3.0 VC	10001
10010	10010 000000 **vsubuwm** v2.03 VX	1/010 000001 **bcdus.** v3.0 VX	10010 000010 **vavguw** v2.03 VX	10010 000011 **vabsduw** v3.0 VX	10010 000100 **vor** v2.03 VX		.0010 000110 **vcmpequw[.]** v2.03 VC	.0010 000111 **vcmpnew[.]** v3.0 VC	10010
10011	10011 000000 **vsubudm** v2.07 VX	1.011 000001 **bcds.** v3.0 VX			10011 000100 **vxor** v2.03 VX		.0011 000110 **vcmpeqfp[.]** v2.03 VC	.0011 000111 **vcmpequd[.]** v2.07 VC	10011
10100	10100 000000 **vsubuqm** v2.07 VX	1.100 000001 **bcdtrunc.** v3.0 VX	10100 000010 **vavgsb** v2.03 VX		10100 000100 **vnor** v2.03 VX			.0100 000111 **vcmpnezb[.]** v3.0 VC	10100
10101	10101 000000 **vsubcuq** v2.07 VX	1/101 000001 **bcdutrunc.** v3.0 VX	10101 000010 **vavgsh** v2.03 VX		10101 000100 **vorc** v2.07 VX			.0101 000111 **vcmpnezh[.]** v3.0 VC	10101
10110	10110 000000 **vsubcuw** v2.03 VX	10110 000001 **XPND004-1A** {expanded}	10110 000010 **vavgsw** v2.03 VX		10110 000100 **vnand** v2.07 VX			.0110 000111 **vcmpnezw[.]** v3.0 VC	10110
10111		1.111 000001 **bcdsr.** v3.0 VX			10111 000100 **vsld** v2.07 VX		.0111 000110 **vcmpgefp[.]** v2.03 VC	.0111 000111 **vcmpequq[.]** v3.1 VC	10111
11000	11000 000000 **vsububs** v2.03 VX	1.000 000001 **bcdadd.** v2.07 VX	11000 000010 **XPND004-2** {expanded}		11000 000100 **mfvscr** v2.03 VX		.1000 000110 **vcmpgtub[.]** v2.03 VC		11000
11001	11001 000000 **vsubuhs** v2.03 VX	1.001 000001 **bcdsub.** v2.07 VX	11001 000010 **XPND004-3** {expanded}		11001 000100 **mtvscr** v2.03 VX		.1001 000110 **vcmpgtuh[.]** v2.03 VC		11001
11010	11010 000000 **vsubuws** v2.03 VX	1/010 000001 **bcdus.** {invalid}	11010 000010 **vshasigmaw** v2.07 VX		11010 000100 **veqv** v2.07 VX		.1010 000110 **vcmpgtuw[.]** v2.03 VC	.1010 000111 **vcmpgtuq[.]** v3.1 VC	11010
11011		1.011 000001 **bcds.** v3.0 VX	11011 000010 **vshasigmad** v2.07 VX		11011 000100 **vsrd** v2.07 VX		.1011 000110 **vcmpgtfp[.]** v2.03 VC	.1011 000111 **vcmpgtud[.]** v2.07 VC	11011
11100	11100 000000 **vsubsbs** v2.03 VX	1.100 000001 **bcdtrunc.** v3.0 VX	11100 000010 **vclzb** v2.07 VX	11100 000011 **vpopcntb** v2.07 VX	11100 000100 **vsrv** v3.0 VX		.1100 000110 **vcmpgtsb[.]** v2.03 VC		11100
11101	11101 000000 **vsubshs** v2.03 VX	1/101 000001 **bcdutrunc.** {invalid}	11101 000010 **vclzh** v2.07 VX	11101 000011 **vpopcnth** v2.07 VX	11101 000100 **vslv** v3.0 VX		.1101 000110 **vcmpgtsh[.]** v2.03 VC		11101
11110	11110 000000 **vsubsws** v2.03 VX	11110 000001 **XPND004-1B** {expanded}	11110 000010 **vclzw** v2.07 VX	11110 000011 **vpopcntw** v2.07 VX	11110 000100 **vclzdm** v3.1 VX		.1110 000110 **vcmpgtsw[.]** v2.03 VC	.1110 000111 **vcmpgtsq[.]** v3.1 VC	11110
11111		1.111 000001 **bcdsr.** v3.0 VX	11111 000010 **vclzd** v2.07 VX	11111 000011 **vpopcntd** v2.07 VX	11111 000100 **vctzdm** v3.1 VX		.1111 000110 **vcmpbfp[.]** v2.03 VC	.1111 000111 **vcmpgtsd[.]** v2.07 VC	11111
	000000	000001	000010	000011	000100	000101	000110	000111	

Table D.5: EXT004: Extended Opcode Map for Opcode Space 0, Primary Opcode 4 (bits 21:31) (cont'd)

	001000	001001	001010	001011	001100	001101	001110	001111	
00000	00000 001000 **vmuloub** v2.03 VX		00000 001010 **vaddfp** v2.03 VX	00000 001011 **vdivuq** v3.1 VX	00000 001100 **vmrghb** v2.03 VX	00000 001101 **XPND004-4A** {expanded}	00000 001110 **vpkuhum** v2.03 VX	00000 001111 **vinsbvlx** v3.1 VX	00000
00001	00001 001000 **vmulouh** v2.03 VX		00001 001010 **vsubfp** v2.03 VX		00001 001100 **vmrghh** v2.03 VX		00001 001110 **vpkuwum** v2.03 VX	00001 001111 **vinshvlx** v3.1 VX	00001
00010	00010 001000 **vmulouw** v2.07 VX	00010 001001 **vmuluwm** v2.07 VX		00010 001011 **vdivuw** v3.1 VX	00010 001100 **vmrghw** v2.03 VX		00010 001110 **vpkuhus** v2.03 VX	00010 001111 **vinswvlx** v3.1 VX	00010
00011	00011 001000 **vmuloud** v3.1 VX			00011 001011 **vdivud** v3.1 VX			00011 001110 **vpkuwus** v2.03 VX	00011 001111 **vinsw** v3.1 VX	00011
00100	00100 001000 **vmulosb** v2.03 VX		00100 001010 **vrefp** v2.03 VX	00100 001011 **vdivsq** v3.1 VX	00100 001100 **vmrglb** v2.03 VX		00100 001110 **vpkshus** v2.03 VX	00100 001111 **vinsbvrx** v3.1 VX	00100
00101	00101 001000 **vmulosh** v2.03 VX		00101 001010 **vrsqrtefp** v2.03 VX		00101 001100 **vmrglh** v2.03 VX		00101 001110 **vpkswus** v2.03 VX	00101 001111 **vinshvrx** v3.1 VX	00101
00110	00110 001000 **vmulosw** v2.07 VX		00110 001010 **vexptefp** v2.03 VX	00110 001011 **vdivsw** v3.1 VX	00110 001100 **vmrglw** v2.03 VX	00110 001101 **vclrlb** v3.1 VX	00110 001110 **vpkshss** v2.03 VX	00110 001111 **vinswvrx** v3.1 VX	00110
00111	00111 001000 **vmulosd** v3.1 VX	00111 001001 **vmulld** v3.1 VX	00111 001010 **vlogefp** v2.03 VX	00111 001011 **vdivsd** v3.1 VX		00111 001101 **vclrrb** v3.1 VX	00111 001110 **vpkswss** v2.03 VX	00111 001111 **vinsd** v3.1 VX	00111
01000	01000 001000 **vmuleub** v2.03 VX		01000 001010 **vrfin** v2.03 VX	01000 001011 **vdiveuq** v3.1 VX	01000 001100 **vspltb** v2.03 VX	01000 001101 **vextractub** v3.0 VX	01000 001110 **vupkhsb** v2.03 VX	01000 001111 **vinsblx** v3.1 VX	01000
01001	01001 001000 **vmuleuh** v2.03 VX		01001 001010 **vrfiz** v2.03 VX		01001 001100 **vsplth** v2.03 VX	01001 001101 **vextractuh** v3.0 VX	01001 001110 **vupkhsh** v2.03 VX	01001 001111 **vinshlx** v3.1 VX	01001
01010	01010 001000 **vmuleuw** v2.07 VX	01010 001001 **vmulhuw** v3.1 VX	01010 001010 **vrfip** v2.03 VX	01010 001011 **vdiveuw** v3.1 VX	01010 001100 **vspltw** v2.03 VX	01010 001101 **vextractuw** v3.0 VX	01010 001110 **vupklsb** v2.03 VX	01010 001111 **vinswlx** v3.1 VX	01010
01011	01011 001000 **vmuleud** v3.1 VX	01011 001001 **vmulhud** v3.1 VX	01011 001010 **vrfim** v2.03 VX	01011 001011 **vdiveud** v3.1 VX		01011 001101 **vextractd** v3.0 VX	01011 001110 **vupklsh** v2.03 VX	01011 001111 **vinsdlx** v3.1 VX	01011
01100	01100 001000 **vmulesb** v2.03 VX		01100 001010 **vcfux** v2.03 VX	01100 001011 **vdivesq** v3.1 VX	01100 001100 **vspltisb** v2.03 VX	01100 001101 **vinsertb** v3.0 VX	01100 001110 **vpkpx** v2.03 VX	01100 001111 **vinsbrx** v3.1 VX	01100
01101	01101 001000 **vmulesh** v2.03 VX		01101 001010 **vcfsx** v2.03 VX		01101 001100 **vspltish** v2.03 VX	01101 001101 **vinserth** v3.0 VX	01101 001110 **vupkhpx** v2.03 VX	01101 001111 **vinshrx** v3.1 VX	01101
01110	01110 001000 **vmulesw** v2.07 VX	01110 001001 **vmulhsw** v3.1 VX	01110 001010 **vctuxs** v2.03 VX	01110 001011 **vdivesw** v3.1 VX	01110 001100 **vspltisw** v2.03 VX	01110 001101 **vinsertw** v3.0 VX		01110 001111 **vinswrx** v3.1 VX	01110
01111	01111 001000 **vmulesd** v3.1 VX	01111 001001 **vmulhsd** v3.1 VX	01111 001010 **vctsxs** v2.03 VX	01111 001011 **vdivesd** v3.1 VX		01111 001101 **vinsertd** v3.0 VX	01111 001110 **vupklpx** v2.03 VX	01111 001111 **vinsdrx** v3.1 VX	01111
10000	10000 001000 **vpmsumb** v2.07 VX		10000 001010 **vmaxfp** v2.03 VX		10000 001100 **vslo** v2.03 VX	10000 001101 **XPND004-4B** {expanded}			10000
10001	10001 001000 **vpmsumh** v2.07 VX		10001 001010 **vminfp** v2.03 VX		10001 001100 **vsro** v2.03 VX		10001 001110 **vpkudum** v2.07 VX		10001
10010	10010 001000 **vpmsumw** v2.07 VX								10010
10011	10011 001000 **vpmsumd** v2.07 VX				10011 001100 **vgnb** v3.1 VX		10011 001110 **vpkudus** v2.07 VX		10011
10100	10100 001000 **vcipher** v2.07 VX	10100 001001 **vcipherlast** v2.07 VX			10100 001100 **vgbbd** v2.07 VX				10100
10101	10101 001000 **vncipher** v2.07 VX	10101 001001 **vncipherlast** v2.07 VX			10101 001100 **vbpermq** v2.07 VX	10101 001101 **vcfuged** v3.1 VX	10101 001110 **vpksdus** v2.07 VX		10101
10110						10110 001101 **vpextd** v3.1 VX			10110
10111	10111 001000 **vsbox** v2.07 VX				10111 001100 **vbpermd** v3.0 VX	10111 001101 **vpdepd** v3.1 VX	10111 001110 **vpksdss** v2.07 VX		10111
11000	11000 001000 **vsum4ubs** v2.03 VX			11000 001011 **vmoduq** v3.1 VX		11000 001101 **vextublx** v3.0 VX			11000
11001	11001 001000 **vsum4shs** v2.03 VX					11001 001101 **vextuhlx** v3.0 VX	11001 001110 **vupkhsw** v2.07 VX		11001
11010	11010 001000 **vsum2sws** v2.03 VX			11010 001011 **vmoduw** v3.1 VX	11010 001100 **vmrgow** v2.07 VX	11010 001101 **vextuwlx** v3.0 VX			11010
11011				11011 001011 **vmodud** v3.1 VX			11011 001110 **vupklsw** v2.07 VX		11011
11100	11100 001000 **vsum4sbs** v2.03 VX			11100 001011 **vmodsq** v3.1 VX		11100 001101 **vextubrx** v3.0 VX			11100
11101						11101 001101 **vextuhrx** v3.0 VX			11101
11110	11110 001000 **vsumsws** v2.03 VX			11110 001011 **vmodsw** v3.1 VX	11110 001100 **vmrgew** v2.07 VX	11110 001101 **vextuwrx** v3.0 VX			11110
11111				11111 001011 **vmodsd** v3.1 VX					11111
	001000	001001	001010	001011	001100	001101	001110	001111	

Table D.5: EXT004: Extended Opcode Map for Opcode Space 0, Primary Opcode 4 (bits 21:31) (cont'd)

	010000	010001	010010	010011	010100	010101	010110	010111	
00000					..... 01010 I **mtvsrbmi** v3.1 DX		00... 010110 I **vsldbi** v3.1 VN	..... 010111 I **vmsumcud** v3.1 VA	00000
00001									00001
00010									00010
00011									00011
00100									00100
00101									00101
00110									00110
00111									00111
01000									01000
01001									01001
01010									01010
01011									01011
01100									01100
01101									01101
01110									01110
01111									01111
10000									10000
10001									10001
10010									10010
10011									10011
10100									10100
10101									10101
10110									10110
10111									10111
11000									11000
11001									11001
11010									11010
11011									11011
11100									11100
11101									11101
11110									11110
11111									11111
	010000	010001	010010	010011	010100	010101	010110	010111	

Table D.5: EXT004: Extended Opcode Map for Opcode Space 0, Primary Opcode 4 (bits 21:31) (cont'd)

	011000	011001	011010	011011	011100	011101	011110	011111	
00000	..... 011000 I **vextdubvlx** v3.1 VA	..... 011001 I **vextdubvrx** v3.1 VA	..... 011010 I **vextduhvlx** v3.1 VA	..... 011011 I **vextduhvrx** v3.1 VA	..... 011100 I **vextduwvlx** v3.1 VA	..... 011101 I **vextduwvrx** v3.1 VA	..... 011110 I **vextddvlx** v3.1 VA	..... 011111 I **vextddvrx** v3.1 VA	00000
00001									00001
00010									00010
00011									00011
00100									00100
00101									00101
00110									00110
00111									00111
01000									01000
01001									01001
01010									01010
01011									01011
01100									01100
01101									01101
01110									01110
01111									01111
10000									10000
10001									10001
10010									10010
10011									10011
10100									10100
10101									10101
10110									10110
10111									10111
11000									11000
11001									11001
11010									11010
11011									11011
11100									11100
11101									11101
11110									11110
11111									11111
	011000	011001	011010	011011	011100	011101	011110	011111	

Table D.5: EXT004: Extended Opcode Map for Opcode Space 0, Primary Opcode 4 (bits 21:31) (cont'd)

	100000	100001	100010	100011	100100	100101	100110	100111	
00000	.... 100000 **vmhaddshs** v2.03 VA	.... 100001 **vmhraddshs** v2.03 VA	.... 100010 **vmladduhm** v2.03 VA	.... 100011 **vmsumudm** v3.0B VA	.... 100100 **vmsumubm** v2.03 VA	.... 100101 **vmsummbm** v2.03 VA	.... 100110 **vmsumuhm** v2.03 VA	.... 100111 **vmsumuhs** v2.03 VA	00000
00001									00001
00010									00010
00011									00011
00100									00100
00101									00101
00110									00110
00111									00111
01000									01000
01001									01001
01010									01010
01011									01011
01100									01100
01101									01101
01110									01110
01111									01111
10000									10000
10001									10001
10010									10010
10011									10011
10100									10100
10101									10101
10110									10110
10111									10111
11000									11000
11001									11001
11010									11010
11011									11011
11100									11100
11101									11101
11110									11110
11111									11111
	100000	100001	100010	100011	100100	100101	100110	100111	

Table D.5: EXT004: Extended Opcode Map for Opcode Space 0, Primary Opcode 4 (bits 21:31) (cont'd)

	101000	101001	101010	101011	101100	101101	101110	101111	
00000	.... 101000 **vmsumshm** v2.03 VA	.... 101001 **vmsumshs** v2.03 VA	.... 101010 **vsel** v2.03 VA	.... 101011 **vperm** v2.03 VA	/.... 101100 **vsldoi** v2.03 VA	.... 101101 **vpermxor** v2.07 VA	.... 101110 **vmaddfp** v2.03 VA	.... 101111 **vnmsubfp** v2.03 VA	00000
00001									00001
00010									00010
00011									00011
00100									00100
00101									00101
00110									00110
00111									00111
01000									01000
01001									01001
01010									01010
01011									01011
01100									01100
01101									01101
01110									01110
01111									01111
10000					/.... 101100 **vsldoi** {invalid}				10000
10001									10001
10010									10010
10011									10011
10100									10100
10101									10101
10110									10110
10111									10111
11000									11000
11001									11001
11010									11010
11011									11011
11100									11100
11101									11101
11110									11110
11111									11111
	101000	101001	101010	101011	101100	101101	101110	101111	

Table D.5: EXT004: Extended Opcode Map for Opcode Space 0, Primary Opcode 4 (bits 21:31) (cont'd)

	110000	110001	110010	110011	110100	110101	110110	110111	
00000	..... 110000 I **maddhd** v3.0 VA	..... 110001 I **maddhdu** v3.0 VA		..... 110011 I **maddld** v3.0 VA					00000
00001									00001
00010									00010
00011									00011
00100									00100
00101									00101
00110									00110
00111									00111
01000									01000
01001									01001
01010									01010
01011									01011
01100									01100
01101									01101
01110									01110
01111									01111
10000									10000
10001									10001
10010									10010
10011									10011
10100									10100
10101									10101
10110									10110
10111									10111
11000									11000
11001									11001
11010									11010
11011									11011
11100									11100
11101									11101
11110									11110
11111									11111
	110000	110001	110010	110011	110100	110101	110110	110111	

Table D.5: EXT004: Extended Opcode Map for Opcode Space 0, Primary Opcode 4 (bits 21:31) (cont'd)

	111000	111001	111010	111011	111100	111101	111110	111111	
00000				..... 111011 | **vpermr** v3.0 VA	..... 111100 | **vaddeuqm** v2.07 VA	..... 111101 | **vaddecuq** v2.07 VA	..... 111110 | **vsubeuqm** v2.07 VA	..... 111111 | **vsubecuq** v2.07 VA	00000
00001									00001
00010									00010
00011									00011
00100									00100
00101									00101
00110									00110
00111									00111
01000									01000
01001									01001
01010									01010
01011									01011
01100									01100
01101									01101
01110									01110
01111									01111
10000									10000
10001									10001
10010									10010
10011									10011
10100									10100
10101									10101
10110									10110
10111									10111
11000									11000
11001									11001
11010									11010
11011									11011
11100									11100
11101									11101
11110									11110
11111									11111
	111000	111001	111010	111011	111100	111101	111110	111111	

Table D.6: EXT006: Extended Opcode Map for Opcode Space 0, Primary Opcode 6 (bits 28:31)

	00	01	10	11	
00	0000 **lxvp** v3.1 DQ	0001 **stxvp** v3.1 DQ			00
01					01
10					10
11					11
	00	01	10	11	

Table D.7: EXT017: Extended Opcode Map for Opcode Space 0, Primary Opcode 17 (bits 30:31)

	00	01	10	11	
		01 **scv** v3.0 SC	1/ **sc** PPC SC	1/ **sc** {invalid}	
	00	01	10	11	

Table D.8: EXT030: Extended Opcode Map for Opcode Space 0, Primary Opcode 30 (bits 27:30)

	00	01	10	11	
00	000. **rldicl[.]** PPC MD	000. **rldicl[.]** PPC MD	001. **rldicr[.]** PPC MD	001. **rldicr[.]** PPC MD	00
01	010. **rldic[.]** PPC MD	010. **rldic[.]** PPC MD	011. **rldimi[.]** PPC MD	011. **rldimi[.]** PPC MD	01
10	1000 **rldcl[.]** PPC MDS	1001 **rldcr[.]** PPC MDS			10
11	1100 {reserved}	1101 {reserved}	1110 {reserved}	1111 {reserved}	11
	00	01	10	11	

Table D.9: EXT057: Extended Opcode Map for Opcode Space 0, Primary Opcode 57 (bits 30:31)

	00	01	10	11	
	00 **lfdp** v2.05 DS	01 {reserved}	10 **lxsd** v3.0 DS	11 **lxssp** v3.0 DS	
	00	01	10	11	

Table D.10: EXT058: Extended Opcode Map for Opcode Space 0, Primary Opcode 58 (bits 30:31)

	00	01	10	11	
	00 **ld** PPC DS	01 **ldu** PPC DS	10 **lwa** PPC DS	11 {reserved}	
	00	01	10	11	

Table D.11: EXT061: Extended Opcode Map for Opcode Space 0, Primary Opcode 61 (bits 29:31)

	00	01	10	11	
0	.00 **stfdp** v2.05 DS	001 **lxv** v3.0 DQ	.10 **stxsd** v3.0 DS	.11 **stxssp** v3.0 DS	0
1		101 **stxv** v3.0 DQ			1
	00	01	10	11	

Table D.12: EXT062: Extended Opcode Map for Opcode Space 0, Primary Opcode 62 (bits 30:31)

	00	01	10	11	
	00 **std** PPC DS	01 **stdu** PPC DS	10 **stq** v2.03 DS	11 {reserved}	
	00	01	10	11	

Table D.13: EXT019: Extended Opcode Map for Opcode Space 0, Primary Opcode 19 (bits 21:30)

	00000	00001	00010	00011	00100	00101	00110	00111	
00000	00000 00000 I **mcrf** P1 XL		..... 00010 I **addpcis** v3.0 DX						00000
00001		00001 00001 I **crnor** P1 XL							00001
00010									00010
00011									00011
00100		00100 00001 I **crandc** P1 XL							00100
00101									00101
00110		00110 00001 I **crxor** P1 XL							00110
00111		00111 00001 I **crnand** P1 XL							00111
01000		01000 00001 I **crand** P1 XL							01000
01001		01001 00001 I **creqv** P1 XL							01001
01010									01010
01011									01011
01100									01100
01101		01101 00001 I **crorc** P1 XL							01101
01110		01110 00001 I **cror** P1 XL							01110
01111									01111
10000									10000
10001									10001
10010									10010
10011									10011
10100									10100
10101									10101
10110									10110
10111									10111
11000									11000
11001									11001
11010									11010
11011									11011
11100									11100
11101									11101
11110									11110
11111									11111
	00000	00001	00010	00011	00100	00101	00110	00111	

Table D.13: EXT019: Extended Opcode Map for Opcode Space 0, Primary Opcode 19 (bits 21:30) (cont'd)

	01000	01001	01010	01011	01100	01101	01110	01111	
00000									00000
00001									00001
00010									00010
00011									00011
00100									00100
00101									00101
00110									00110
00111									00111
01000									01000
01001									01001
01010									01010
01011									01011
01100									01100
01101									01101
01110									01110
01111									01111
10000									10000
10001									10001
10010									10010
10011									10011
10100									10100
10101									10101
10110									10110
10111									10111
11000									11000
11001									11001
11010									11010
11011									11011
11100									11100
11101									11101
11110									11110
11111									11111
	01000	01001	01010	01011	01100	01101	01110	01111	

Table D.13: EXT019: Extended Opcode Map for Opcode Space 0, Primary Opcode 19 (bits 21:30) (cont'd)

	10000	10001	10010	10011	10100	10101	10110	10111	
00000	00000 10000 I **bclr[l]** P1 XL		00000 10010 III **rfid** PPC XL						00000
00001			00001 10010 {reserved} XL	00001 10011 {reserved}					00001
00010			00010 10010 III **rfscv** v3.0 XL						00010
00011									00011
00100			00100 10010 I **rfebb** v2.07 XL				00100 10110 II **isync** P1 XL		00100
00101									00101
00110									00110
00111									00111
01000			01000 10010 III **hrfid** v2.02 XL						01000
01001			01001 10010 III **urfid** v3.0C XL						01001
01010									01010
01011			01011 10010 III **stop** v3.0 XL						01011
01100			01100 10010 {reserved}						01100
01101			01101 10010 {reserved}						01101
01110			01110 10010 {reserved}						01110
01111			01111 10010 {reserved}						01111
10000	10000 10000 I **bcctrl** P1 XL								10000
10001	10001 10000 I **bctarl** v2.07 XL								10001
10010									10010
10011									10011
10100									10100
10101									10101
10110									10110
10111									10111
11000									11000
11001									11001
11010									11010
11011									11011
11100									11100
11101									11101
11110									11110
11111									11111
	10000	10001	10010	10011	10100	10101	10110	10111	

Table D.13: EXT019: Extended Opcode Map for Opcode Space 0, Primary Opcode 19 (bits 21:30) (cont'd)

	11000	11001	11010	11011	11100	11101	11110	11111	
00000									00000
00001									00001
00010									00010
00011									00011
00100									00100
00101									00101
00110									00110
00111									00111
01000									01000
01001									01001
01010									01010
01011									01011
01100									01100
01101									01101
01110									01110
01111									01111
10000									10000
10001									10001
10010									10010
10011									10011
10100									10100
10101									10101
10110									10110
10111									10111
11000									11000
11001									11001
11010									11010
11011									11011
11100									11100
11101									11101
11110									11110
11111									11111
	11000	11001	11010	11011	11100	11101	11110	11111	

Table D.14: EXT031: Extended Opcode Map for Opcode Space 0, Primary Opcode 31 (bits 21:30)

	00000	00001	00010	00011	00100	00101	00110	00111	
00000	00000 00000 I **cmp** P1 X				00000 00100 I **tw** P1 X		00000 00110 I **lvsl** v2.03 X	00000 00111 I **lvebx** v2.03 X	00000
00001	00001 00000 I **cmpl** P1 X				....1 00100 {reserved}		00001 00110 I **lvsr** v2.03 X	00001 00111 I **lvehx** v2.03 X	00001
00010	00010 00000 {reserved}				00010 00100 I **td** PPC X			00010 00111 I **lvewx** v2.03 X	00010
00011					....1 00100 {reserved}			00011 00111 I **lvx** v2.03 X	00011
00100	00100 00000 I **setb** v3.0 V			00100 00011 {reserved}				00100 00111 I **stvebx** v2.03 X	00100
00101				00101 00011 {reserved}	....1 00100 {reserved}			00101 00111 I **stvehx** v2.03 X	00101
00110	00110 00000 I **cmprb** v3.0 X							00110 00111 I **stvewx** v2.03 X	00110
00111	00111 00000 I **cmpeqb** v3.0 X				....1 00100 {reserved}			00111 00111 I **stvx** v2.03 X	00111
01000							01000 00110 {reserved}		01000
01001					....1 00100 {reserved}				01001
01010				01010 00011 {reserved}					01010
01011					....1 00100 {reserved}			01011 00111 I **lvxl** v2.03 X	01011
01100	01100 00000 I **setbc** v3.1 X								01100
01101	01101 00000 I **setbcr** v3.1 X				....1 00100 {reserved}				01101
01110	01110 00000 I **setnbc** v3.1 X			01110 00011 {reserved}			01110 00110 {reserved}		01110
01111	01111 00000 I **setnbcr** v3.1 X				....1 00100 {reserved}		01111 00110 {reserved}	01111 00111 I **stvxl** v2.03 X	01111
10000	10000 00000 {reserved}							1.000 00111 {reserved}	10000
10001	10001 00000 {reserved}				....1 00100 {reserved}			1.001 00111 {reserved}	10001
10010	10010 00000 I **mcrxrx** v3.0 X						10010 00110 II **lwat** v3.0 X		10010
10011					....1 00100 {reserved}		10011 00110 II **ldat** v3.0 X		10011
10100								1.100 00111 {reserved}	10100
10101					....1 00100 {reserved}			1.101 00111 {reserved}	10101
10110							10110 00110 II **stwat** v3.0 X		10110
10111					....1 00100 {reserved}		10111 00110 II **stdat** v3.0 X		10111
11000							11000 00110 II **copy** v3.0 X	1.000 00111 {reserved}	11000
11001					....1 00100 {reserved}			1.001 00111 {reserved}	11001
11010							11010 00110 II **cpabort** v3.0 X		11010
11011					....1 00100 {reserved}				11011
11100							11100 00110 II **paste[.]** v3.0 X	1.100 00111 {reserved}	11100
11101					....1 00100 {reserved}			1.101 00111 {reserved}	11101
11110							11110 00110 {reserved}		11110
11111					....1 00100 {reserved}		11111 00110 {reserved}		11111
	00000	00001	00010	00011	00100	00101	00110	00111	

Table D.14: EXT031: Extended Opcode Map for Opcode Space 0, Primary Opcode 31 (bits 21:30) (cont'd)

	01000	01001	01010	01011	01100	01101	01110	01111	
00000	00000 01000 I **subfc[.]** P1 XO	/0000 01001 I **mulhdu[.]** PPC XO	00000 01010 I **addc[.]** P1 XO	/0000 01011 I **mulhwu[.]** PPC XO	00000 01100 I **lxsiwzx** v2.07 X	00000 01101 I **lxvrbx** v3.1 X		..... 01111 I **isel** v2.03 A	00000
00001	00001 01000 I **subf[.]** PPC XO					00001 01101 I **lxvrhx** v3.1 X			00001
00010		/0010 01001 I **mulhd[.]** PPC XO	/0010 01010 I **addg6s** v2.06 XO	/0010 01011 I **mulhw[.]** PPC XO	00010 01100 I **lxsiwax** v2.07 X	00010 01101 I **lxvrwx** v3.1 X	00010 01110 III **msgsndu** v3.0C X		00010
00011	00011 01000 I **neg[.]** P1 XO			00011 01011 {reserved}		00011 01101 I **lxvrdx** v3.1 X	00011 01110 III **msgclru** v3.0C X		00011
00100	00100 01000 I **subfe[.]** P1 XO		00100 01010 I **adde[.]** P1 XO		00100 01100 I **stxsiwx** v2.07 X	00100 01101 I **stxvrbx** v3.1 X	00100 01110 III **msgsndp** v2.07 X		00100
00101			..101 01010 I **addex** v3.0B Z23			00101 01101 I **stxvrhx** v3.1 X	00101 01110 III **msgclrp** v2.07 X		00101
00110	00110 01000 I **subfze[.]** P1 XO		00110 01010 I **addze[.]** P1 XO			00110 01101 I **stxvrwx** v3.1 X	00110 01110 III **msgsnd** v2.07 X		00110
00111	00111 01000 I **subfme[.]** P1 XO	00111 01001 I **mulld[.]** PPC XO	00111 01010 I **addme[.]** P1 XO	00111 01011 I **mullw[.]** P1 XO		00111 01101 I **stxvrdx** v3.1 X	00111 01110 III **msgclr** v2.07 X		00111
01000	01000 01000 {reserved}	01000 01001 I **modud** v3.0 X	01000 01010 I **add[.]** P1 XO	01000 01011 I **moduw** v3.0 X	0100/ 01100 I **lxvx** v3.0 X	01000 01101 I **lxvl** v3.0 X			01000
01001					0100/ 01100 **lxvx** {invalid}	01001 01101 I **lxvll** v3.0 X	01001 01110 I **mfbhrbe** v2.07 XFX		01001
01010				01010 01011 {reserved}	01010 01100 I **lxvdsx** v2.06 X	01010 01101 I **lxvpx** v3.1 X			01010
01011	01011 01000 {reserved}			01011 01011 {reserved}	01011 01100 I **lxvwsx** v3.0 X				01011
01100		01100 01001 I **divdeu[.]** v2.06 XO		01100 01011 I **divweu[.]** v2.06 XO	01100 01100 I **stxvx** v3.0 X	01100 01101 I **stxvl** v3.0 X			01100
01101		01101 01001 I **divde[.]** v2.06 XO	..101 01010 I **addex** v3.0B Z23	01101 01011 I **divwe[.]** v2.06 XO		01101 01101 I **stxvll** v3.0 X	01101 01110 I **clrbhrb** v2.07 X		01101
01110		01110 01001 I **divdu[.]** PPC XO		01110 01011 I **divwu[.]** PPC XO		01110 01101 I **stxvpx** v3.1 X			01110
01111	01111 01000 {reserved}	01111 01001 I **divd[.]** PPC XO		01111 01011 I **divw[.]** PPC XO					01111
10000	10000 01000 I **subfco[.]** P1 XO	/0000 01001 **mulhdu[.]** {invalid}	10000 01010 I **addco[.]** P1 XO	/0000 01011 **mulhwu[.]** {invalid}	10000 01100 I **lxsspx** v2.07 X				10000
10001	10001 01000 I **subfo[.]** PPC XO								10001
10010		/0010 01001 **mulhd[.]** {invalid}	/0010 01010 **addg6s** {invalid}	/0010 01011 **mulhw[.]** {invalid}	10010 01100 I **lxsdx** v2.06 X				10010
10011	10011 01000 I **nego[.]** P1 XO			10011 01011 {reserved}					10011
10100	10100 01000 I **subfeo[.]** P1 XO		10100 01010 I **addeo[.]** P1 XO		10100 01100 I **stxsspx** v2.07 X		10100 01110 II **tbegin.** v2.07 {reserved} X		10100
10101			..101 01010 I **addex** v3.0B Z23				10101 01110 II **tend.** v2.07 {reserved} X		10101
10110	10110 01000 I **subfzeo[.]** P1 XO		10110 01010 I **addzeo[.]** P1 XO		10110 01100 I **stxsdx** v2.06 X		10110 01110 II **tcheck** v2.07 {reserved} X		10110
10111	10111 01000 I **subfmeo[.]** P1 XO	10111 01001 I **mulldo[.]** PPC XO	10111 01010 I **addmeo[.]** P1 XO	10111 01011 I **mullwo[.]** P1 XO			10111 01110 II **tsr.** v2.07 {reserved} X		10111
11000	11000 01000 {reserved}	11000 01001 I **modsd** v3.0 X	11000 01010 I **addo[.]** P1 XO	11000 01011 I **modsw** v3.0 X	11000 01100 I **lxvw4x** v2.06 X	11000 01101 I **lxsibzx** v3.0 X	11000 01110 II **tabortwc.** v2.07 {reserved} X		11000
11001					11001 01100 I **lxvh8x** v3.0 X	11001 01101 I **lxsihzx** v3.0 X	11001 01110 II **tabortdc.** v2.07 {reserved} X		11001
11010				11010 01011 {reserved}	11010 01100 I **lxvd2x** v2.06 X		11010 01110 II **tabortwci.** v2.07 {reserved} X		11010
11011	11011 01000 {reserved}			11011 01011 {reserved}	11011 01100 I **lxvb16x** v3.0 X		11011 01110 II **tabortdci.** v2.07 {reserved} X		11011
11100		11100 01001 I **divdeuo[.]** v2.06 XO		11100 01011 I **divweuo[.]** v2.06 XO	11100 01100 I **stxvw4x** v2.06 X	11100 01101 I **stxsibx** v3.0 X	11100 01110 II **tabort.** v2.07 {reserved} X		11100
11101		11101 01001 I **divdeo[.]** v2.06 XO	..101 01010 I **addex** v3.0B Z23	11101 01011 I **divweo[.]** v2.06 XO	11101 01100 I **stxvh8x** v3.0 X	11101 01101 I **stxsihx** v3.0 X	11101 01110 III **treclaim.** v2.07 {reserved} X		11101
11110		11110 01001 I **divduo[.]** PPC XO		11110 01011 I **divwuo[.]** PPC XO	11110 01100 I **stxvd2x** v2.06 X				11110
11111	11111 01000 {reserved}	11111 01001 I **divdo[.]** PPC XO		11111 01011 I **divwo[.]** PPC XO	11111 01100 I **stxvb16x** v3.0 X		11111 01110 III **trechkpt.** v2.07 {reserved} X		11111
	01000	01001	01010	01011	01100	01101	01110	01111	

Table D.14: EXT031: Extended Opcode Map for Opcode Space 0, Primary Opcode 31 (bits 21:30) (cont'd)

	10000	10001	10010	10011	10100	10101	10110	10111	
00000				00000 10011 I **mfcr/mfocrf** P1/v2.01 XFX	00000 10100 II **lwarx** PPC X	00000 10101 I **ldx** PPC X	00000 10110 II **icbt** v2.07 X	00000 10111 I **lwzx** P1 X	00000
00001				00001 10011 I **mfvsrd** v2.07 X	00001 10100 II **lbarx** v2.06 X	00001 10101 I **ldux** PPC X	00001 10110 II **dcbst** PPC X	00001 10111 I **lwzux** P1 X	00001
00010			00010 10010 {reserved}	00010 10011 III **mfmsr** P1 X	00010 10100 II **ldarx** PPC X		00010 10110 II **dcbf** PPC X	00010 10111 I **lbzx** P1 X	00010
00011			00011 10010 {reserved}	00011 10011 I **mfvsrwz** v2.07 X	00011 10100 II **lharx** v2.06 X		00011 10110 {reserved}	00011 10111 I **lbzux** P1 X	00011
00100	00100 10000 I **mtcrf/mtocrf** P1/v2.01 XFX		00100 10010 III **mtmsr** P1 X	00100 10011 {reserved}		00100 10101 I **stdx** PPC X	00100 10110 II **stwcx.** PPC X	00100 10111 I **stwx** P1 X	00100
00101		00101 10001 **XPND031** {expanded}	00101 10010 III **mtmsrd** PPC X	00101 10011 I **mtvsrd** v2.07 X		00101 10101 I **stdux** PPC X	00101 10110 I **stqcx.** v2.07 X	00101 10111 I **stwux** P1 X	00101
00110			00110 10010 III {reserved}	00110 10011 I **mtvsrwa** v2.07 X			00110 10110 II **stdcx.** PPC X	00110 10111 I **stbx** P1 X	00110
00111			00111 10010 III {reserved}	00111 10011 I **mtvsrwz** v2.07 X			00111 10110 II **dcbtst** PPC X	00111 10111 I **stbux** P1 X	00111
01000			01000 10010 III **tlbiel** v2.03 X		01000 10100 I **lqarx** v2.07 X	01000 10101 {reserved}	01000 10110 II **dcbt** PPC X	01000 10111 I **lhzx** P1 X	01000
01001			01001 10010 III **tlbie** P1 X	01001 10011 I **mfvsrld** v3.0 X	01001 10100 {reserved}		01001 10110 {reserved}	01001 10111 I **lhzux** P1 X	01001
01010			01010 10010 III **slbsync** v3.0 X	01010 10011 X **mfspr** P1 XFX		01010 10101 I **lwax** PPC X	01010 10110 {reserved}	01010 10111 I **lhax** P1 X	01010
01011			01011 10010 {reserved}	01011 10011 II **mftb** PPC XFX		01011 10101 I **lwaux** PPC X	01011 10110 {reserved}	01011 10111 I **lhaux** P1 X	01011
01100			01100 10010 III **slbmte** v2.00 X	01100 10011 I **mtvsrws** v3.0 X				01100 10111 I **sthx** P1 X	01100
01101			01101 10010 III **slbie** PPC X	01101 10011 I **mtvsrdd** v3.0 X			01101 10110 {reserved}	01101 10111 I **sthux** P1 X	01101
01110			01110 10010 III **slbieg** v3.0 X	01110 10011 X **mtspr** P1 XFX			01110 10110 {reserved}		01110
01111			01111 10010 III **slbia** PPC X	01111 10011 {reserved}			01111 10110 {reserved}		01111
10000			10000 10010 {reserved-nop}	10000 10011 {reserved}	10000 10100 I **ldbrx** v2.06 X	10000 10101 I **lswx** P1 X	10000 10110 I **lwbrx** P1 X	10000 10111 I **lfsx** P1 X	10000
10001			10001 10010 {reserved-nop}			10001 10101 {reserved}	10001 10110 III **tlbsync** PPC X	10001 10111 I **lfsux** P1 X	10001
10010			10010 10010 {reserved-nop}	10010 10011 {reserved}		10010 10101 I **lswi** P1 X	10010 10110 II **sync** P1 X	10010 10111 I **lfdx** P1 X	10010
10011			10011 10010 {reserved-nop}	10011 10011 {reserved}		10011 10101 {reserved}	10011 10110 {reserved}	10011 10111 I **lfdux** P1 X	10011
10100			10100 10010 III **hashstp** v3.1B X	10100 10011 {reserved}	10100 10100 I **stdbrx** v2.06 X	10100 10101 I **stswx** P1 X	10100 10110 I **stwbrx** P1 X	10100 10111 I **stfsx** P1 X	10100
10101			10101 10010 III **hashchkp** v3.1B X			10101 10101 {reserved}	10101 10110 II **stbcx.** v2.06 X	10101 10111 I **stfsux** P1 X	10101
10110			10110 10010 I **hashst** v3.1B X			10110 10101 I **stswi** P1 X	10110 10110 II **sthcx.** v2.06 X	10110 10111 I **stfdx** P1 X	10110
10111			10111 10010 I **hashchk** v3.1B X	10111 10011 I **darn** v3.0 X		10111 10101 {reserved}	10111 10110 {reserved}	10111 10111 I **stfdux** P1 X	10111
11000						11000 10101 III **lwzcix** v2.05 X	11000 10110 I **lhbrx** P1 X	11000 10111 I **lfdpx** v2.05 X	11000
11001			11001 10010 {reserved}			11001 10101 III **lhzcix** v2.05 X	11001 10110 {reserved}	11001 10111 {reserved}	11001
11010			11010 10010 III **slbiag** v3.0B X	11010 10011 III **slbmfev** v2.00 X		11010 10101 III **lbzcix** v2.05 X	11010 10110 II **eieio** PPC X	11010 10111 I **lfiwax** v2.05 X	11010
11011						11011 10101 III **ldcix** v2.05 X	11011 10110 III **msgsync** v3.0 X	11011 10111 I **lfiwzx** v2.06 X	11011
11100			11100 10010 {reserved}	11100 10011 III **slbmfee** v2.00 X		11100 10101 III **stwcix** v2.05 X	11100 10110 I **sthbrx** P1 X	11100 10111 I **stfdpx** v2.05 X	11100
11101			11101 10010 {reserved}			11101 10101 III **sthcix** v2.05 X		11101 10111 {reserved}	11101
11110			11110 10010 {reserved}	11110 10011 III **slbfee.** v2.05 X		11110 10101 III **stbcix** v2.05 X	11110 10110 II **icbi** PPC X	11110 10111 I **stfiwx** PPC X	11110
11111			11111 10010 {reserved}			11111 10101 III **stdcix** v2.05 X	11111 10110 II **dcbz** P1 X		11111
	10000	10001	10010	10011	10100	10101	10110	10111	

Table D.14: EXT031: Extended Opcode Map for Opcode Space 0, Primary Opcode 31 (bits 21:30) (cont'd)

	11000	11001	11010	11011	11100	11101	11110	11111	
00000	00000 11000 I **slw[.]** P1 X		00000 11010 I **cntlzw[.]** P1 X	00000 11011 I **sld[.]** PPC X	00000 11100 I **and[.]** P1 X	00000 11101 {reserved}	00000 11110 II **wait** v2.03 X		00000
00001			00001 11010 I **cntlzd[.]** PPC X	00001 11011 I **cntlzdm** v3.1 X	00001 11100 I **andc[.]** P1 X	00001 11101 {reserved}			00001
00010						00010 11101 {reserved}			00010
00011			00011 11010 I **popcntb** v2.02 X		00011 11100 I **nor[.]** P1 X				00011
00100	00100 11000 {reserved}	00100 11001 {reserved}	00100 11010 I **prtyw** v2.05 X	00100 11011 I **brw** v3.1 X	00100 11100 I **pdepd** v3.1 X				00100
00101	00101 11000 {reserved}		00101 11010 I **prtyd** v2.05 X	00101 11011 I **brd** v3.1 X	00101 11100 I **pextd** v3.1 X				00101
00110	00110 11000 {reserved}	00110 11001 {reserved}		00110 11011 I **brh** v3.1 X	00110 11100 I **cfuged** v3.1 X				00110
00111	00111 11000 {reserved}				00111 11100 I **bpermd** v2.06 X				00111
01000			01000 11010 I **cdtbcd** v2.06 X		01000 11100 I **eqv[.]** P1 X				01000
01001			01001 11010 I **cbcdtd** v2.06 X		01001 11100 I **xor[.]** P1 X				01001
01010									01010
01011			01011 11010 I **popcntw** v2.06 X						01011
01100					01100 11100 I **orc[.]** P1 X				01100
01101					01101 11100 I **or[.]** P1 X				01101
01110					01110 11100 I **nand[.]** P1 X				01110
01111			01111 11010 I **popcntd** v2.06 X		01111 11100 I **cmpb** v2.05 X				01111
10000	10000 11000 I **srw[.]** P1 X	10000 11001 {reserved}	10000 11010 I **cnttzw[.]** v3.0 X	10000 11011 I **srd[.]** PPC X		10000 11101 {reserved}			10000
10001			10001 11010 I **cnttzd[.]** v3.0 X	10001 11011 I **cnttzdm** v3.1 X					10001
10010									10010
10011									10011
10100	10100 11000 {reserved}	10100 11001 {reserved}							10100
10101	10101 11000 {reserved}								10101
10110	10110 11000 {reserved}	10110 11001 {reserved}							10110
10111	10111 11000 {reserved}								10111
11000	11000 11000 I **sraw[.]** P1 X		11000 11010 I **srad[.]** PPC X						11000
11001	11001 11000 I **srawi[.]** P1 X		11001 1101. I **sradi[.]** PPC XS	11001 1101. I **sradi[.]** PPC XS					11001
11010									11010
11011			11011 1101. I **extswsli[.]** v3.0 XS	11011 1101. I **extswsli[.]** v3.0 XS					11011
11100	11100 11000 {reserved}	11100 11001 {reserved}	11100 11010 I **extsh[.]** P1 X						11100
11101	11101 11000 {reserved}		11101 11010 I **extsb[.]** PPC X						11101
11110			11110 11010 I **extsw[.]** PPC X						11110
11111									11111
	11000	11001	11010	11011	11100	11101	11110	11111	

Table D.15: EXT059: Extended Opcode Map for Opcode Space 0, Primary Opcode 59 (bits 21:30)

	00000	00001	00010	00011	00100	00101	00110	00111	
00000			00000 00010 **dadd[.]** v2.05 X	..000 00011 **dqua[.]** v2.05 Z23					00000
00001			00001 00010 **dmul[.]** v2.05 X	..001 00011 **drrnd[.]** v2.05 Z23					00001
00010			.0010 00010 **dscli[.]** v2.05 Z22	..010 00011 **dquai[.]** v2.05 Z23					00010
00011			.0011 00010 **dscri[.]** v2.05 Z22	..011 00011 **drintx[.]** v2.05 Z23					00011
00100			00100 00010 **dcmpo** v2.05 X						00100
00101			00101 00010 **dtstex** v2.05 X						00101
00110			.0110 00010 **dtstdc** v2.05 Z22						00110
00111			.0111 00010 **dtstdg** v2.05 Z22	..111 00011 **drintn[.]** v2.05 Z23					00111
01000			01000 00010 **dctdp[.]** v2.05 X	..000 00011 **dqua[.]** v2.05 Z23					01000
01001			01001 00010 **dctfix[.]** v2.05 X	..001 00011 **drrnd[.]** v2.05 Z23					01001
01010			01010 00010 **ddedpd[.]** v2.05 X	..010 00011 **dquai[.]** v2.05 Z23					01010
01011			01011 00010 **dxex[.]** v2.05 X	..011 00011 **drintx[.]** v2.05 Z23					01011
01100									01100
01101									01101
01110									01110
01111				..111 00011 **drintn[.]** v2.05 Z23					01111
10000			10000 00010 **dsub[.]** v2.05 X	..000 00011 **dqua[.]** v2.05 Z23					10000
10001			10001 00010 **ddiv[.]** v2.05 X	..001 00011 **drrnd[.]** v2.05 Z23					10001
10010			.0010 00010 **dscli[.]** v2.05 Z22	..010 00011 **dquai[.]** v2.05 Z23					10010
10011			.0011 00010 **dscri[.]** v2.05 Z22	..011 00011 **drintx[.]** v2.05 Z23					10011
10100			10100 00010 **dcmpu** v2.05 X						10100
10101			10101 00010 **dtstsf** v2.05 X	10101 00011 **dtstsfi** v3.0 X					10101
10110			.0110 00010 **dtstdc** v2.05 Z22						10110
10111			.0111 00010 **dtstdg** v2.05 Z22	..111 00011 **drintn[.]** v2.05 Z23					10111
11000			11000 00010 **drsp[.]** v2.05 X	..000 00011 **dqua[.]** v2.05 Z23					11000
11001			11001 00010 **dcffix[.]** v2.06 X	..001 00011 **drrnd[.]** v2.05 Z23					11001
11010			11010 00010 **denbcd[.]** v2.05 X	..010 00011 **dquai[.]** v2.05 Z23					11010
11011			11011 00010 **diex[.]** v2.05 X	..011 00011 **drintx[.]** v2.05 Z23					11011
11100									11100
11101									11101
11110									11110
11111				..111 00011 **drintn[.]** v2.05 Z23					11111
	00000	00001	00010	00011	00100	00101	00110	00111	

Table D.15: EXT059: Extended Opcode Map for Opcode Space 0, Primary Opcode 59 (bits 21:30) (cont'd)

	01000	01001	01010	01011	01100	01101	01110	01111	
00000	00000 010.. **[pm]xvi8ger4pp** v3.1 [MMIRR:]XX3				00000 011.. **[pm]xvi8ger4** v3.1 [MMIRR:]XX3				00000
00001									00001
00010	00010 010.. **[pm]xvf16ger2pp** v3.1 [MMIRR:]XX3				00010 011.. **[pm]xvf16ger2** v3.1 [MMIRR:]XX3				00010
00011	00011 010.. **[pm]xvf32gerpp** v3.1 [MMIRR:]XX3				00011 011.. **[pm]xvf32ger** v3.1 [MMIRR:]XX3				00011
00100	00100 010.. **[pm]xvi4ger8pp** v3.1 [MMIRR:]XX3				00100 011.. **[pm]xvi4ger8** v3.1 [MMIRR:]XX3				00100
00101	00101 010.. **[pm]xvi16ger2spp** v3.1 [MMIRR:]XX3				00101 011.. **[pm]xvi16ger2s** v3.1 [MMIRR:]XX3				00101
00110	00110 010.. **[pm]xvbf16ger2pp** v3.1 [MMIRR:]XX3				00110 011.. **[pm]xvbf16ger2** v3.1 [MMIRR:]XX3				00110
00111	00111 010.. **[pm]xvf64gerpp** v3.1 [MMIRR:]XX3				00111 011.. **[pm]xvf64ger** v3.1 [MMIRR:]XX3				00111
01000									01000
01001					01001 011.. **[pm]xvi16ger2** v3.1 [MMIRR:]XX3				01001
01010	01010 010.. **[pm]xvf16ger2np** v3.1 [MMIRR:]XX3								01010
01011	01011 010.. **[pm]xvf32gernp** v3.1 [MMIRR:]XX3								01011
01100					01100 011.. **[pm]xvi8ger4spp** v3.1 [MMIRR:]XX3				01100
01101					01101 011.. **[pm]xvi16ger2pp** v3.1 [MMIRR:]XX3				01101
01110	01110 010.. **[pm]xvbf16ger2np** v3.1 [MMIRR:]XX3								01110
01111	01111 010.. **[pm]xvf64gernp** v3.1 [MMIRR:]XX3								01111
10000									10000
10001									10001
10010	10010 010.. **[pm]xvf16ger2pn** v3.1 [MMIRR:]XX3								10010
10011	10011 010.. **[pm]xvf32gerpn** v3.1 [MMIRR:]XX3								10011
10100									10100
10101									10101
10110	10110 010.. **[pm]xvbf16ger2pn** v3.1 [MMIRR:]XX3								10110
10111	10111 010.. **[pm]xvf64gerpn** v3.1 [MMIRR:]XX3								10111
11000									11000
11001									11001
11010	11010 010.. **[pm]xvf16ger2nn** v3.1 [MMIRR:]XX3						11010 01110 **fcfids[.]** v2.06 X		11010
11011	11011 010.. **[pm]xvf32gernn** v3.1 [MMIRR:]XX3								11011
11100									11100
11101									11101
11110	11110 010.. **[pm]xvbf16ger2nn** v3.1 [MMIRR:]XX3						11110 01110 **fcfidus[.]** v2.06 X		11110
11111	11111 010.. **[pm]xvf64gernn** v3.1 [MMIRR:]XX3								11111
	01000	01001	01010	01011	01100	01101	01110	01111	

Table D.15: EXT059: Extended Opcode Map for Opcode Space 0, Primary Opcode 59 (bits 21:30) (cont'd)

	10000	10001	10010	10011	10100	10101	10110	10111	
00000			///// 10010 I **fdivs[.]** PPC A		///// 10100 I **fsubs[.]** PPC A	///// 10101 I **fadds[.]** PPC A	///// 10110 I **fsqrts[.]** PPC A		00000
00001			///// 10010 **fdivs[.]** {invalid}		///// 10100 **fsubs[.]** {invalid}	///// 10101 **fadds[.]** {invalid}	///// 10110 **fsqrts[.]** {invalid}		00001
00010									00010
00011									00011
00100									00100
00101									00101
00110									00110
00111									00111
01000									01000
01001									01001
01010									01010
01011									01011
01100									01100
01101									01101
01110									01110
01111									01111
10000									10000
10001									10001
10010									10010
10011									10011
10100									10100
10101									10101
10110									10110
10111									10111
11000									11000
11001									11001
11010									11010
11011									11011
11100									11100
11101									11101
11110									11110
11111									11111
	10000	10001	10010	10011	10100	10101	10110	10111	

Table D.15: EXT059: Extended Opcode Map for Opcode Space 0, Primary Opcode 59 (bits 21:30) (cont'd)

	11000	11001	11010	11011	11100	11101	11110	11111	
00000	///// 11000 **fres[.]** PPC A	..... 11001 **fmuls[.]** PPC A	///// 11010 **frsqrtes[.]** v2.02 A		..... 11100 **fmsubs[.]** PPC A	..... 11101 **fmadds[.]** PPC A	..... 11110 **fnmsubs[.]** PPC A	..... 11111 **fnmadds[.]** PPC A	00000
00001	///// 11000 **fres[.]** {invalid}		///// 11010 **frsqrtes[.]** {invalid}						00001
00010									00010
00011									00011
00100									00100
00101									00101
00110									00110
00111									00111
01000									01000
01001									01001
01010									01010
01011									01011
01100									01100
01101									01101
01110									01110
01111									01111
10000									10000
10001									10001
10010									10010
10011									10011
10100									10100
10101									10101
10110									10110
10111									10111
11000									11000
11001									11001
11010									11010
11011									11011
11100									11100
11101									11101
11110									11110
11111									11111
	11000	11001	11010	11011	11100	11101	11110	11111	

Table D.16: EXT060: Extended Opcode Map for Opcode Space 0, Primary Opcode 60 (bits 21:30)

	00000	00001	00010	00011	00100	00101	00110	00111	
00000	00000 000.. **xsaddsp** v2.07 XX3				00000 001.. **xsmaddasp** v2.07 XX3				00000
00001	00001 000.. **xssubsp** v2.07 XX3				00001 001.. **xsmaddmsp** v2.07 XX3				00001
00010	00010 000.. **xsmulsp** v2.07 XX3				00010 001.. **xsmsubasp** v2.07 XX3				00010
00011	00011 000.. **xsdivsp** v2.07 XX3				00011 001.. **xsmsubmsp** v2.07 XX3				00011
00100	00100 000.. **xsadddp** v2.06 XX3				00100 001.. **xsmaddadp** v2.06 XX3				00100
00101	00101 000.. **xssubdp** v2.06 XX3				00101 001.. **xsmaddmdp** v2.06 XX3				00101
00110	00110 000.. **xsmuldp** v2.06 XX3				00110 001.. **xsmsubadp** v2.06 XX3				00110
00111	00111 000.. **xsdivdp** v2.06 XX3				00111 001.. **xsmsubmdp** v2.06 XX3				00111
01000	01000 000.. **xvaddsp** v2.06 XX3				01000 001.. **xvmaddasp** v2.06 XX3				01000
01001	01001 000.. **xvsubsp** v2.06 XX3				01001 001.. **xvmaddmsp** v2.06 XX3				01001
01010	01010 000.. **xvmulsp** v2.06 XX3				01010 001.. **xvmsubasp** v2.06 XX3				01010
01011	01011 000.. **xvdivsp** v2.06 XX3				01011 001.. **xvmsubmsp** v2.06 XX3				01011
01100	01100 000.. **xvadddp** v2.06 XX3				01100 001.. **xvmaddadp** v2.06 XX3				01100
01101	01101 000.. **xvsubdp** v2.06 XX3				01101 001.. **xvmaddmdp** v2.06 XX3				01101
01110	01110 000.. **xvmuldp** v2.06 XX3				01110 001.. **xvmsubadp** v2.06 XX3				01110
01111	01111 000.. **xvdivdp** v2.06 XX3				01111 001.. **xvmsubmdp** v2.06 XX3				01111
10000	10000 000.. **xsmaxcdp** v3.0 XX3				10000 001.. **xsnmaddasp** v2.07 XX3				10000
10001	10001 000.. **xsmincdp** v3.0 XX3				10001 001.. **xsnmaddmsp** v2.07 XX3				10001
10010	10010 000.. **xsmaxjdp** v3.0 XX3				10010 001.. **xsnmsubasp** v2.07 XX3				10010
10011	10011 000.. **xsminjdp** v3.0 XX3				10011 001.. **xsnmsubmsp** v2.07 XX3				10011
10100	10100 000.. **xsmaxdp** v2.06 XX3				10100 001.. **xsnmaddadp** v2.06 XX3				10100
10101	10101 000.. **xsmindp** v2.06 XX3				10101 001.. **xsnmaddmdp** v2.06 XX3				10101
10110	10110 000.. **xscpsgndp** v2.06 XX3				10110 001.. **xsnmsubadp** v2.06 XX3				10110
10111					10111 001.. **xsnmsubmdp** v2.06 XX3				10111
11000	11000 000.. **xvmaxsp** v2.06 XX3				11000 001.. **xvnmaddasp** v2.06 XX3				11000
11001	11001 000.. **xvminsp** v2.06 XX3				11001 001.. **xvnmaddmsp** v2.06 XX3				11001
11010	11010 000.. **xvcpsgnsp** v2.06 XX3				11010 001.. **xvnmsubasp** v2.06 XX3				11010
11011	11011 000.. **xviexpsp** v3.0 XX3				11011 001.. **xvnmsubmsp** v2.06 XX3				11011
11100	11100 000.. **xvmaxdp** v2.06 XX3				11100 001.. **xvnmaddadp** v2.06 XX3				11100
11101	11101 000.. **xvmindp** v2.06 XX3				11101 001.. **xvnmaddmdp** v2.06 XX3				11101
11110	11110 000.. **xvcpsgndp** v2.06 XX3				11110 001.. **xvnmsubadp** v2.06 XX3				11110
11111	11111 000.. **xviexpdp** v3.0 XX3				11111 001.. **xvnmsubmdp** v2.06 XX3				11111
	00000	00001	00010	00011	00100	00101	00110	00111	

Table D.16: EXT060: Extended Opcode Map for Opcode Space 0, Primary Opcode 60 (bits 21:30) (cont'd)

	01000	01001	01010	01011	01100	01101	01110	01111	
00000	0..00 010.. **xxsldwi** v2.06 XX3				00000 011.. **xscmpeqdp** v3.0 XX3				00000
00001	0..01 010.. **xxpermdi** v2.06 XX3				00001 011.. **xscmpgtdp** v3.0 XX3				00001
00010	00010 010.. **xxmrghw** v2.06 XX3				00010 011.. **xscmpgedp** v3.0 XX3				00010
00011	00011 010.. **xxperm** v3.0 XX3								00011
00100	0..00 010.. **xxsldwi** v2.06 XX3				00100 011.. **xscmpudp** v2.06 XX3				00100
00101	0..01 010.. **xxpermdi** v2.06 XX3				00101 011.. **xscmpodp** v2.06 XX3				00101
00110	00110 010.. **xxmrglw** v2.06 XX3								00110
00111	00111 010.. **xxpermr** v3.0 XX3				00111 011.. **xscmpexpdp** v3.0 XX3				00111
01000	0..00 010.. **xxsldwi** v2.06 XX3				01000 011.. **xvcmpeqsp** v2.06 XX3				01000
01001	0..01 010.. **xxpermdi** v2.06 XX3				01001 011.. **xvcmpgtsp** v2.06 XX3				01001
01010	01010 0100. **xxspltw** v2.06 XX2		01010 0101. **xxextractuw** v3.0 XX2		01010 011.. **xvcmpgesp** v2.06 XX3				01010
01011	01011 01000 **XPND060-1** {expanded}		01011 0101. **xxinsertw** v3.0 XX2						01011
01100	0..00 010.. **xxsldwi** v2.06 XX3				01100 011.. **xvcmpeqdp** v2.06 XX3				01100
01101	0..01 010.. **xxpermdi** v2.06 XX3				01101 011.. **xvcmpgtdp** v2.06 XX3				01101
01110					01110 011.. **xvcmpgedp** v2.06 XX3				01110
01111									01111
10000	10000 010.. **xxland** v2.06 XX3								10000
10001	10001 010.. **xxlandc** v2.06 XX3								10001
10010	10010 010.. **xxlor** v2.06 XX3								10010
10011	10011 010.. **xxlxor** v2.06 XX3								10011
10100	10100 010.. **xxlnor** v2.06 XX3								10100
10101	10101 010.. **xxlorc** v2.07 XX3								10101
10110	10110 010.. **xxlnand** v2.07 XX3								10110
10111	10111 010.. **xxleqv** v2.07 XX3								10111
11000					11000 011.. **xvcmpeqsp.** v2.06 XX3				11000
11001					11001 011.. **xvcmpgtsp.** v2.06 XX3				11001
11010					11010 011.. **xvcmpgesp.** v2.06 XX3				11010
11011									11011
11100					11100 011.. **xvcmpeqdp.** v2.06 XX3				11100
11101					11101 011.. **xvcmpgtdp.** v2.06 XX3				11101
11110					11110 011.. **xvcmpgedp.** v2.06 XX3				11110
11111									11111
	01000	01001	01010	01011	01100	01101	01110	01111	

Table D.16: EXT060: Extended Opcode Map for Opcode Space 0, Primary Opcode 60 (bits 21:30) (cont'd)

	10000	10001	10010	10011	10100	10101	10110	10111	
00000					00000 1010. **xsrsqrtesp** v2.07 XX2		00000 1011. **xssqrtsp** v2.07 XX2		00000
00001					00001 1010. **xsresp** v2.07 XX2				00001
00010									00010
00011									00011
00100	00100 1000. **xscvdpuxws** v2.06 XX2		00100 1001. **xsrdpi** v2.06 XX2		00100 1010. **xsrsqrtedp** v2.06 XX2		00100 1011. **xssqrtdp** v2.06 XX2		00100
00101	00101 1000. **xscvdpsxws** v2.06 XX2		00101 1001. **xsrdpiz** v2.06 XX2		00101 1010. **xsredp** v2.06 XX2				00101
00110			00110 1001. **xsrdpip** v2.06 XX2		00110 1010. **xstsqrtdp** v2.06 XX2		00110 1011. **xsrdpic** v2.06 XX2		00110
00111			00111 1001. **xsrdpim** v2.06 XX2		00111 101.. **xstdivdp** v2.06 XX3				00111
01000	01000 1000. **xvcvspuxws** v2.06 XX2		01000 1001. **xvrspi** v2.06 XX2		01000 1010. **xvrsqrtesp** v2.06 XX2		01000 1011. **xvsqrtsp** v2.06 XX2		01000
01001	01001 1000. **xvcvspsxws** v2.06 XX2		01001 1001. **xvrspiz** v2.06 XX2		01001 1010. **xvresp** v2.06 XX2				01001
01010	01010 1000. **xvcvuxwsp** v2.06 XX2		01010 1001. **xvrspip** v2.06 XX2		01010 1010. **xvtsqrtsp** v2.06 XX2		01010 1011. **xvrspic** v2.06 XX2		01010
01011	01011 1000. **xvcvsxwsp** v2.06 XX2		01011 1001. **xvrspim** v2.06 XX2		01011 101.. **xvtdivsp** v2.06 XX3				01011
01100	01100 1000. **xvcvdpuxws** v2.06 XX2		01100 1001. **xvrdpi** v2.06 XX2		01100 1010. **xvrsqrtedp** v2.06 XX2		01100 1011. **xvsqrtdp** v2.06 XX2		01100
01101	01101 1000. **xvcvdpsxws** v2.06 XX2		01101 1001. **xvrdpiz** v2.06 XX2		01101 1010. **xvredp** v2.06 XX2				01101
01110	01110 1000. **xvcvuxwdp** v2.06 XX2		01110 1001. **xvrdpip** v2.06 XX2		01110 1010. **xvtsqrtdp** v2.06 XX2		01110 1011. **xvrdpic** v2.06 XX2		01110
01111	01111 1000. **xvcvsxwdp** v2.06 XX2		01111 1001. **xvrdpim** v2.06 XX2		01111 101.. **xvtdivdp** v2.06 XX3				01111
10000			10000 1001. **xscvdpsp** v2.06 XX2				10000 1011. **xscvdpspn** v2.07 XX2		10000
10001			10001 1001. **xsrsp** v2.07 XX2						10001
10010	10010 1000. **xscvuxdsp** v2.07 XX2				10010 1010. **xststdcsp** v3.0 XX2				10010
10011	10011 1000. **xscvsxdsp** v2.07 XX2								10011
10100	10100 1000. **xscvdpuxds** v2.06 XX2		10100 1001. **xscvspdp** v2.06 XX2				10100 1011. **xscvspdpn** v2.07 XX2		10100
10101	10101 1000. **xscvdpsxds** v2.06 XX2		10101 1001. **xsabsdp** v2.06 XX2				10101 1011 **XPND060-2** {expanded}		10101
10110	10110 1000. **xscvuxddp** v2.06 XX2		10110 1001. **xsnabsdp** v2.06 XX2		10110 1010. **xststdcdp** v3.0 XX2				10110
10111	10111 1000. **xscvsxddp** v2.06 XX2		10111 1001. **xsnegdp** v2.06 XX2						10111
11000	11000 1000. **xvcvspuxds** v2.06 XX2		11000 1001. **xvcvdpsp** v2.06 XX2						11000
11001	11001 1000. **xvcvspsxds** v2.06 XX2		11001 1001. **xvabssp** v2.06 XX2						11001
11010	11010 1000. **xvcvuxdsp** v2.06 XX2		11010 1001. **xvnabssp** v2.06 XX2		1101. 101.. **xvtstdcsp** v3.0 XX2				11010
11011	11011 1000. **xvcvsxdsp** v2.06 XX2		11011 1001. **xvnegsp** v2.06 XX2						11011
11100	11100 1000. **xvcvdpuxds** v2.06 XX2		11100 1001. **xvcvspdp** v2.06 XX2		11100 10100 **xxgenpcvbm** v3.1 X	11100 10101 **xxgenpcvhm** v3.1 X	11100 10110 **xsiexpdp** v3.0 X		11100
11101	11101 1000. **xvcvdpsxds** v2.06 XX2		11101 1001. **xvabsdp** v2.06 XX2		11101 11101 10100 **xxgenpcvwm** v3.1 X	11101 11101 10101 **xxgenpcvdm** v3.1 X	11101 1011. **XPND060-3** {expanded}		11101
11110	11110 1000. **xvcvuxddp** v2.06 XX2		11110 1001. **xvnabsdp** v2.06 XX2		1111. 101.. **xvtstdcdp** v3.0 XX2				11110
11111	11111 1000. **xvcvsxddp** v2.06 XX2		11111 1001. **xvnegdp** v2.06 XX2						11111
	10000	10001	10010	10011	10100	10101	10110	10111	

Table D.16: EXT060: Extended Opcode Map for Opcode Space 0, Primary Opcode 60 (bits 21:30) (cont'd)

	11000	11001	11010	11011	11100	11101	11110	11111	
00000	..... 11... **xxsel** v2.06 XX4								00000
00001									00001
00010									00010
00011									00011
00100									00100
00101									00101
00110									00110
00111									00111
01000									01000
01001									01001
01010									01010
01011									01011
01100									01100
01101									01101
01110									01110
01111									01111
10000									10000
10001									10001
10010									10010
10011									10011
10100									10100
10101									10101
10110									10110
10111									10111
11000									11000
11001									11001
11010									11010
11011									11011
11100									11100
11101									11101
11110									11110
11111									11111
	11000	11001	11010	11011	11100	11101	11110	11111	

Table D.17: EXT063: Extended Opcode Map for Opcode Space 0, Primary Opcode 63 (bits 21:30)

	00000	00001	00010	00011	00100	00101	00110	00111	
00000	00000 00000 **fcmpu** P1 X		00000 00010 **daddq[.]** v2.05 X	..000 00011 **dquaq[.]** v2.05 Z23	00000 00100 **xsaddqp[o]** v3.0 X	..000 00101 **xsrqpi[x]** v3.0 Z23			00000
00001	00001 00000 **fcmpo** P1 X		00001 00010 **dmulq[.]** v2.05 X	..001 00011 **drrndq[.]** v2.05 Z23	00001 00100 **xsmulqp[o]** v3.0 X	..001 00101 **xsrqpxp** v3.0 Z23	00001 00110 **mtfsb1[.]** P1 X		00001
00010	00010 00000 **mcrfs** P1 X		.0010 00010 **dscliq[.]** v2.05 Z22	..010 00011 **dquaiq[.]** v2.05 Z23	00010 00100 **xscmpeqqp** v3.1 X		00010 00110 **mtfsb0[.]** P1 X		00010
00011			.0011 00010 **dscriq[.]** v2.05 Z22	..011 00011 **drintxq[.]** v2.05 Z23	00011 00100 **xscpsgnqp** v3.0 X				00011
00100	00100 00000 **ftdiv** v2.06 X		00100 00010 **dcmpoq** v2.05 X		00100 00100 **xscmpoqp** v3.0 X		00100 00110 **mtfsfi[.]** P1 X		00100
00101	00101 00000 **ftsqrt** v2.06 X		00101 00010 **dtstexq** v2.05 X		00101 00100 **xscmpexpqp** v3.0 X				00101
00110			.0110 00010 **dtstdcq** v2.05 Z22		00110 00100 **xscmpgeqp** v3.1 X				00110
00111			.0111 00010 **dtstdgq** v2.05 Z22	..111 00011 **drintnq[.]** v2.05 Z23	00111 00100 **xscmpgtqp** v3.1 X				00111
01000			01000 00010 **dctqpq[.]** v2.05 X	..000 00011 **dquaq[.]** v2.05 Z23		..000 00101 **xsrqpi[x]** v3.0 Z23			01000
01001			01001 00010 **dctfixq[.]** v2.05 X	..001 00011 **drrndq[.]** v2.05 Z23		..001 00101 **xsrqpxp** v3.0 Z23			01001
01010			01010 00010 **ddedpdq[.]** v2.05 X	..010 00011 **dquaiq[.]** v2.05 Z23					01010
01011			01011 00010 **dxexq[.]** v2.05 X	..011 00011 **drintxq[.]** v2.05 Z23					01011
01100					01100 00100 **xsmaddqp[o]** v3.0 X				01100
01101					01101 00100 **xsmsubqp[o]** v3.0 X				01101
01110					01110 00100 **xsnmaddqp[o]** v3.0 X				01110
01111				..111 00011 **drintnq[.]** v2.05 Z23	01111 00100 **xsnmsubqp[o]** v3.0 X				01111
10000			10000 00010 **dsubq[.]** v2.05 X	..000 00011 **dquaq[.]** v2.05 Z23	10000 00100 **xssubqp[o]** v3.0 X	..000 00101 **xsrqpi[x]** v3.0 Z23			10000
10001			10001 00010 **ddivq[.]** v2.05 X	..001 00011 **drrndq[.]** v2.05 Z23	10001 00100 **xsdivqp[o]** v3.0 X	..001 00101 **xsrqpxp** v3.0 Z23			10001
10010			.0010 00010 **dscliq[.]** v2.05 Z22	..010 00011 **dquaiq[.]** v2.05 Z23				10010 00111 **XPND063-4** {expanded}	10010
10011			.0011 00010 **dscriq[.]** v2.05 Z22	..011 00011 **drintxq[.]** v2.05 Z23					10011
10100			10100 00010 **dcmpuq** v2.05 X		10100 00100 **xscmpuqp** v3.0 X				10100
10101			10101 00010 **dtstsfq** v2.05 X	10101 00011 **dtstsfiq** v3.0 X	10101 00100 **xsmaxcqp** v3.1 X				10101
10110			.0110 00010 **dtstdcq** v2.05 Z22		10110 00100 **xststdcqp** v3.0 X			10110 00111 **mtfsf[.]** P1 XFL	10110
10111			.0111 00010 **dtstdgq** v2.05 Z22	..111 00011 **drintnq[.]** v2.05 Z23	10111 00100 **xsmincqp** v3.1 X				10111
11000			11000 00010 **drdpq[.]** v2.05 X	..000 00011 **dquaq[.]** v2.05 Z23		..000 00101 **xsrqpi[x]** v3.0 Z23			11000
11001			11001 00010 **dcffixq[.]** v2.05 X	..001 00011 **drrndq[.]** v2.05 Z23	11001 00100 **XPND063-2** {expanded}	..001 00101 **xsrqpxp** v3.0 Z23			11001
11010			11010 00010 **denbcdq[.]** v2.05 X	..010 00011 **dquaiq[.]** v2.05 Z23	11010 00100 **XPND063-3** {expanded}		11010 00110 **fmrgow** v2.07 X		11010
11011			11011 00010 **diexq[.]** v2.05 X	..011 00011 **drintxq[.]** v2.05 Z23	11011 00100 **xsiexpqp** v3.0 X				11011
11100									11100
11101									11101
11110							11110 00110 **fmrgew** v2.07 X		11110
11111			11111 00010 **XPND063-1** {expanded}	..111 00011 **drintnq[.]** v2.05 Z23					11111
	00000	00001	00010	00011	00100	00101	00110	00111	

Table D.17: EXT063: Extended Opcode Map for Opcode Space 0, Primary Opcode 63 (bits 21:30) (cont'd)

	01000	01001	01010	01011	01100	01101	01110	01111	
00000	00000 01000 I **fcpsgn[.]** v2.05 X				00000 01100 I **frsp[.]** P1 X		00000 01110 I **fctiw[.]** P2 X	00000 01111 I **fctiwz[.]** P2 X	00000
00001	00001 01000 I **fneg[.]** P1 X								00001
00010	00010 01000 I **fmr[.]** P1 X								00010
00011									00011
00100	00100 01000 I **fnabs[.]** P1 X						00100 01110 I **fctiwu[.]** v2.06 X	00100 01111 I **fctiwuz[.]** v2.06 X	00100
00101									00101
00110									00110
00111									00111
01000	01000 01000 I **fabs[.]** P1 X								01000
01001									01001
01010									01010
01011									01011
01100	01100 01000 I **frin[.]** v2.02 X								01100
01101	01101 01000 I **friz[.]** v2.02 X								01101
01110	01110 01000 I **frip[.]** v2.02 X								01110
01111	01111 01000 I **frim[.]** v2.02 X								01111
10000									10000
10001									10001
10010									10010
10011									10011
10100									10100
10101									10101
10110									10110
10111									10111
11000									11000
11001							11001 01110 I **fctid[.]** PPC X	11001 01111 I **fctidz[.]** PPC X	11001
11010							11010 01110 I **fcfid[.]** PPC X		11010
11011									11011
11100									11100
11101							11101 01110 I **fctidu[.]** v2.06 X	11101 01111 I **fctiduz[.]** v2.06 X	11101
11110							11110 01110 I **fcfidu[.]** v2.06 X		11110
11111									11111
	01000	01001	01010	01011	01100	01101	01110	01111	

Table D.17: EXT063: Extended Opcode Map for Opcode Space 0, Primary Opcode 63 (bits 21:30) (cont'd)

	10000	10001	10010	10011	10100	10101	10110	10111	
00000			///// 10010 **fdiv[.]** P1 A		///// 10100 **fsub[.]** P1 A	///// 10101 **fadd[.]** P1 A	///// 10110 **fsqrt[.]** P2 A	..... 10111 **fsel[.]** PPC A	00000
00001			///// 10010 **fdiv[.]** {invalid}		///// 10100 **fsub[.]** {invalid}	///// 10101 **fadd[.]** {invalid}	///// 10110 **fsqrt[.]** {invalid}		00001
00010									00010
00011									00011
00100									00100
00101									00101
00110									00110
00111									00111
01000									01000
01001									01001
01010									01010
01011									01011
01100									01100
01101									01101
01110									01110
01111									01111
10000									10000
10001									10001
10010									10010
10011									10011
10100									10100
10101									10101
10110									10110
10111									10111
11000									11000
11001									11001
11010									11010
11011									11011
11100									11100
11101									11101
11110									11110
11111									11111
	10000	10001	10010	10011	10100	10101	10110	10111	

Table D.17: EXT063: Extended Opcode Map for Opcode Space 0, Primary Opcode 63 (bits 21:30) (cont'd)

	11000	11001	11010	11011	11100	11101	11110	11111	
00000	///// 11000 **fre[.]** v2.02 A	..... 11001 **fmul[.]** P1 A	///// 11010 **frsqrte[.]** PPC A		..... 11100 **fmsub[.]** P1 A	..... 11101 **fmadd[.]** P1 A	..... 11110 **fnmsub[.]** P1 A	..... 11111 **fnmadd[.]** P1 A	00000
00001	///// 11000 **fre[.]** {invalid}		///// 11010 **frsqrte[.]** {invalid}						00001
00010									00010
00011									00011
00100									00100
00101									00101
00110									00110
00111									00111
01000									01000
01001									01001
01010									01010
01011									01011
01100									01100
01101									01101
01110									01110
01111									01111
10000									10000
10001									10001
10010									10010
10011									10011
10100									10100
10101									10101
10110									10110
10111									10111
11000									11000
11001									11001
11010									11010
11011									11011
11100									11100
11101									11101
11110									11110
11111									11111
	11000	11001	11010	11011	11100	11101	11110	11111	

Table D.18: EXT132: Extended Opcode Map for Opcode Space 1, Primary Opcode 32 (bits 11:15)

	000	001	010	011	100	101	110	111	
00	000.. **xxsplti32dx** v3.1 8RR:D				0010. **xxspltidp** v3.1 8RR:D		0011. **xxspltiw** v3.1 8RR:D		00
01									01
10									10
11									11
	000	001	010	011	100	101	110	111	

Table D.19: EXT133: Extended Opcode Map for Opcode Space 1, Primary Opcode 33 (bits 26:30)

	000	001	010	011	100	101	110	111	
00	00... **xxblendvb** v3.1 8RR:XX4								00
01	01... **xxblendvh** v3.1 8RR:XX4								01
10	10... **xxblendvw** v3.1 8RR:XX4								10
11	11... **xxblendvd** v3.1 8RR:XX4								11
	000	001	010	011	100	101	110	111	

Table D.20: EXT134: Extended Opcode Map for Opcode Space 1, Primary Opcode 34 (bits 26:30)

	000	001	010	011	100	101	110	111	
00	00... **xxpermx** v3.1 8RR:XX4								00
01	01... **xxeval** v3.1 8RR:XX4								01
10									10
11									11
	000	001	010	011	100	101	110	111	

Table D.21: XPND004-1A: Expanded Opcode Map for Instruction 0x10000581 (bits 11:15)

	000	001	010	011	100	101	110	111	
00	00000 1/110 000001 **bcdctsq.** v3.0 VX		00010 1.110 000001 **bcdcfsq.** v3.0 VX		00100 1.110 000001 **bcdctz.** v3.0 VX	00101 1/110 000001 **bcdctn.** v3.0 VX	00110 1.110 000001 **bcdcfz.** v3.0 VX	00111 1.110 000001 **bcdcfn.** v3.0 VX	00
01									01
10									10
11								11111 1.110 000001 **bcdsetsgn.** v3.0 VX	11
	000	001	010	011	100	101	110	111	

Table D.22: XPND004-1B: Expanded Opcode Map for Instruction 0x10000781 (bits 11:15)

	000	001	010	011	100	101	110	111	
00	00000 1/110 000001 **bcdctsq.** {invalid}		00010 1.110 000001 **bcdcfsq.** v3.0 VX		00100 1.110 000001 **bcdctz.** v3.0 VX	00101 1/110 000001 **bcdctn.** {invalid}	00110 1.110 000001 **bcdcfz.** v3.0 VX	00111 1.110 000001 **bcdcfn.** v3.0 VX	00
01									01
10									10
11								11111 1.110 000001 **bcdsetsgn.** v3.0 VX	11
	000	001	010	011	100	101	110	111	

Table D.23: XPND004-2: Expanded Opcode Map for Instruction 0x10000602 (bits 11:15)

	000	001	010	011	100	101	110	111	
00	00000 11000 000010 **vclzlsbb** v3.0 VX	00001 11000 000010 **vctzlsbb** v3.0 VX					00110 11000 000010 **vnegw** v3.0 VX	00111 11000 000010 **vnegd** v3.0 VX	00
01	01000 11000 000010 **vprtybw** v3.0 VX	01001 11000 000010 **vprtybd** v3.0 VX	01010 11000 000010 **vprtybq** v3.0 VX						01
10	10000 11000 000010 **vextsb2w** v3.0 VX	10001 11000 000010 **vextsh2w** v3.0 VX							10
11	11000 11000 000010 **vextsb2d** v3.0 VX	11001 11000 000010 **vextsh2d** v3.0 VX	11010 11000 000010 **vextsw2d** v3.0 VX	11011 11000 000010 **vextsd2q** v3.1 VX	11100 11000 000010 **vctzb** v3.0 VX	11101 11000 000010 **vctzh** v3.0 VX	11110 11000 000010 **vctzw** v3.0 VX	11111 11000 000010 **vctzd** v3.0 VX	11
	000	001	010	011	100	101	110	111	

Table D.24: XPND004-3: Expanded Opcode Map for Instruction 0x1000_0642 (bits 11:15)

	000	001	010	011	100	101	110	111	
00	00000 11001 000010 **vexpandbm** v3.1 VX	00001 11001 000010 **vexpandhm** v3.1 VX	00010 11001 000010 **vexpandwm** v3.1 VX	00011 11001 000010 **vexpanddm** v3.1 VX	00100 11001 000010 **vexpandqm** v3.1 VX				00
01	01000 11001 000010 **vextractbm** v3.1 VX	01001 11001 000010 **vextracthm** v3.1 VX	01010 11001 000010 **vextractwm** v3.1 VX	01011 11001 000010 **vextractdm** v3.1 VX	01100 11001 000010 **vextractqm** v3.1 VX				01
10	10000 11001 000010 **mtvsrbm** v3.1 VX	10001 11001 000010 **mtvsrhm** v3.1 VX	10010 11001 000010 **mtvsrwm** v3.1 VX	10011 11001 000010 **mtvsrdm** v3.1 VX	10100 11001 000010 **mtvsrqm** v3.1 VX				10
11	1100. 11001 000010 **vcntmbb** v3.1 VX		1101. 11001 000010 **vcntmbd** v3.1 VX		1110. 11001 000010 **vcntmbh** v3.1 VX		1111. 11001 000010 **vcntmbw** v3.1 VX		11
	000	001	010	011	100	101	110	111	

Table D.25: XPND004-4A: Expanded Opcode Map for Instruction 0x1000000D (bits 11:15)

	000	001	010	011	100	101	110	111	
00	00000 00000 001101 I **vstribl** v3.1 VC	00001 00000 001101 I **vstribr** v3.1 VC	00010 00000 001101 I **vstrihl** v3.1 VC	00011 00000 001101 I **vstrihr** v3.1 VC					00
01									01
10									10
11									11
	000	001	010	011	100	101	110	111	

Table D.26: XPND004-4B: Expanded Opcode Map for Instruction 0x1000040D (bits 11:15)

	000	001	010	011	100	101	110	111	
00	00000 10000 001101 I **vstribl.** v3.1 VC	00001 10000 001101 I **vstribr.** v3.1 VC	00010 10000 001101 I **vstrihl.** v3.1 VC	00011 10000 001101 I **vstrihr.** v3.1 VC					00
01									01
10									10
11									11
	000	001	010	011	100	101	110	111	

Table D.27: XPND031: Expanded Opcode Map for Instruction 0x7C000162 (bits 11:15)

	000	001	010	011	100	101	110	111	
00	00000 00101 10001 I **xxmfacc** v3.1 X	00001 00101 10001 I **xxmtacc** v3.1 X		00011 00101 10001 I **xxsetaccz** v3.1 X					00
01									01
10									10
11									11
	000	001	010	011	100	101	110	111	

Table D.28: XPND060-1: Expanded Opcode Map for Instruction 0xF000_02D0 (bits 11:15)

	000	001	010	011	100	101	110	111	
00	00... 01011 01000 I **xxspltib** v3.0 X								00
01									01
10									10
11								11111 01011 01000 I **lxvkq** v3.1 X	11
	000	001	010	011	100	101	110	111	

Table D.29: XPND060-2: Expanded Opcode Map for Instruction 0xF000_056C (bits 11:15)

	000	001	010	011	100	101	110	111	
00	00000 10101 1011. I **xsxexpdp** v3.0 XX2	00001 10101 1011. I **xsxsigdp** v3.0 XX2							00
01									01
10	10000 10101 1011. I **xscvhpdp** v3.0 XX2	10001 10101 1011. I **xscvdphp** v3.0 XX2							10
11									11
	000	001	010	011	100	101	110	111	

Table D.30: XPND060-3: Expanded Opcode Map for Instruction 0xF000_076C (bits 11:15)

	000	001	010	011	100	101	110	111	
00	00000 11101 1011. I **xvxexpdp** v3.0 XX2	00001 11101 1011. I **xvxsigdp** v3.0 XX2	00010 11101 1011. I **xvtlsbb** v3.1 XX2					00111 11101 1011. I **xxbrh** v3.0 XX2	00
01	01000 11101 1011. I **xvxexpsp** v3.0 XX2	01001 11101 1011. I **xvxsigsp** v3.0 XX2						01111 11101 1011. I **xxbrw** v3.0 XX2	01
10	10000 11101 1011. I **xvcvbf16spn** v3.1 XX2	10001 11101 1011. I **xvcvspbf16** v3.1 XX2						10111 11101 1011. I **xxbrd** v3.0 XX2	10
11	11000 11101 1011. I **xvcvhpsp** v3.0 XX2	11001 11101 1011. I **xvcvsphp** v3.0 XX2						11111 11101 1011. I **xxbrq** v3.0 XX2	11
	000	001	010	011	100	101	110	111	

Table D.31: XPND063-1: Expanded Opcode Map for Instruction 0xFC00_07C4 (bits 11:15)

	000	001	010	011	100	101	110	111	
00	00000 11111 00010 **dcffixqq** v3.1 X	00001 11111 00010 **dctfixqq** v3.1 X							00
01									01
10									10
11									11
	000	001	010	011	100	101	110	111	

Table D.32: XPND063-2: Expanded Opcode Map for Instruction 0xFC00_0648 (bits 11:15)

	000	001	010	011	100	101	110	111	
00	00000 11001 00100 **xsabsqp** v3.0 X		00010 11001 00100 **xsxexpqp** v3.0 X						00
01	01000 11001 00100 **xsnabsqp** v3.0 X								01
10	10000 11001 00100 **xsnegqp** v3.0 X		10010 11001 00100 **xsxsigqp** v3.0 X						10
11				11011 11001 00100 **xssqrtqp[o]** v3.0 X					11
	000	001	010	011	100	101	110	111	

Table D.33: XPND063-3: Expanded Opcode Map for Instruction 0xFC00_0688 (bits 11:15)

	000	001	010	011	100	101	110	111	
00	00000 11010 00100 **xscvqpuqz** v3.1 X	00001 11010 00100 **xscvqpuwz** v3.0 X	00010 11010 00100 **xscvudqp** v3.0 X	00011 11010 00100 **xscvuqqp** v3.1 X					00
01	01000 11010 00100 **xscvqpsqz** v3.1 X	01001 11010 00100 **xscvqpswz** v3.0 X	01010 11010 00100 **xscvsdqp** v3.0 X	01011 11010 00100 **xscvsqqp** v3.1 X					01
10		10001 11010 00100 **xscvqpudz** v3.0 X			10100 11010 00100 **xscvqpdp[o]** v3.0 X		10110 11010 00100 **xscvdpqp** v3.0 X		10
11		11001 11010 00100 **xscvqpsdz** v3.0 X							11
	000	001	010	011	100	101	110	111	

Table D.34: XPND063-4: Expanded Opcode Map for Instruction 0xFC00_048E (bits 11:15)

	000	001	010	011	100	101	110	111	
00	00000 10010 00111 **mffs[.]** P1 X	00001 10010 00111 **mffsce** v3.0B X							00
01									01
10					10100 10010 00111 **mffscdrn** v3.0B X	10101 10010 00111 **mffscdrni** v3.0B X	10110 10010 00111 **mffscrn** v3.0B X	10111 10010 00111 **mffscrni** v3.0B X	10
11	11000 10010 00111 **mtffsl** v3.0B X								11
	000	001	010	011	100	101	110	111	

Appendix E. Power ISA Instruction Set Sorted by Opcode

This appendix lists all the instructions in the Power ISA, sorted by primary opcode (bits 0-5), then by extended opcode column (bits 26:31, if any), then by extended opcode row (bits 21:25, if any), then by expanded opcode (bits 11:15, if any). The encoding of the values in the columns is described below.

Instruction

/	Instruction bit that corresponds to a reserved field, must have a value of 0, otherwise invalid form.
.	Instruction bit that corresponds to an operand bit, may have a value of either 0 or 1.
0	Instruction bit that corresponds to an opcode bit having a value 0.
1	Instruction bit that corresponds to an opcode bit having a value 1.

OpenPOWER Compliancy Subsets

X...	Instruction included in the Scalar Fixed-Point Compliancy subset
.X..	Instruction included in the Scalar Fixed-Point + Floating-Point Compliancy subset.
..X.	Instruction included in the Linux Compliancy subset.
...X	Instruction included in the AIX Compliancy subset.

Linux Optional Category

AMO	Instruction part of Atomic Memory Operations category.
BFP128	Instruction part of Quad-Precision Floating-Point category.
BHRB	Instruction part of Branch History Rolling Buffer category.
DFP	Instruction part of Decimal Floating-Point category.
EBB	Instruction part of Event-Based Branch category.
MMA	Instruction part of Matrix-Multiplication Assist category.

Always Optional Category

MMA	Instruction part of Matrix-Multiplication Assist category.

Version

P1	Instruction introduced in POWER Architecture.
P2	Instruction introduced in POWER2 Architecture.
PPC	Instruction introduced in PowerPC Architecture prior to v2.00.
v2.00	Instruction introduced in PowerPC Architecture Version 2.00.
v2.01	Instruction introduced in PowerPC Architecture Version 2.01.
v2.02	Instruction introduced in PowerPC Architecture Version 2.02.
v2.03	Instruction introduced in Power ISA Version 2.03.
v2.04	Instruction introduced in Power ISA Version 2.04.
v2.05	Instruction introduced in Power ISA Version 2.05.
v2.06	Instruction introduced in Power ISA Version 2.06.
v2.07	Instruction introduced in Power ISA Version 2.07.
v3.0	Instruction introduced in Power ISA Version 3.0.
v3.0B	Instruction introduced in Power ISA Version 3.0B.
v3.0C	Instruction introduced in Power ISA Version 3.0C.
v3.1	Instruction introduced in Power ISA Version 3.1.
v3.1B	Instruction introduced in Power ISA Version 3.1B.

Privilege

P	Denotes an instruction that is treated as privileged.
O	Denotes an instruction that is treated as privileged or nonprivileged (or hypervisor-privileged for mtspr), depending on the SPR or PMR number.
PI	Denotes an instruction that is illegal in privileged state.
HV	Denotes an instruction that can be executed only in hypervisor state.
UV	Denotes an instruction that can be executed only in ultravisor state.

Mode Dependency

Except as described below and in Section 1.10.3, “Effective Address Calculation”, in Book I, all instructions are independent of whether the processor is in 32-bit or 64-bit mode.

CT If the instruction tests the Count Register, it tests the low-order 32 bits in 32-bit mode and all 64 bits in 64-bit mode.

SR The setting of status registers (such as XER and CR0) is mode-dependent.

64 The instruction can be executed only in 64-bit mode.

Table E.1: Power ISA Instruction Set Sorted by Opcode

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
000010	I	..XX			tdi	PPC			95	Trap Doubleword Immediate D-form
000011	I	XXXX			twi	P1			94	Trap Word Immediate D-form
000100 .... 00000 000000	I	..XX			vaddubm	v2.03			315	Vector Add Unsigned Byte Modulo VX-form
000100 .... 00001 000000	I	..XX			vadduhm	v2.03			315	Vector Add Unsigned Halfword Modulo VX-form
000100 .... 00010 000000	I	..XX			vadduwm	v2.03			316	Vector Add Unsigned Word Modulo VX-form
000100 .... 00011 000000	I	..XX			vaddudm	v2.07			316	Vector Add Unsigned Doubleword Modulo VX-form
000100 .... 00100 000000	I	..XX			vadduqm	v2.07			319	Vector Add Unsigned Quadword Modulo VX-form
000100 .... 00101 000000	I	..XX			vaddcuq	v2.07			320	Vector Add & write Carry-out Unsigned Quadword VX-form
000100 .... 00110 000000	I	..XX			vaddcuw	v2.03			313	Vector Add & write Carry-out Unsigned Word VX-form
000100 .... 01000 000000	I	..XX			vaddubs	v2.03			317	Vector Add Unsigned Byte Saturate VX-form
000100 .... 01001 000000	I	..XX			vadduhs	v2.03			317	Vector Add Unsigned Halfword Saturate VX-form
000100 .... 01010 000000	I	..XX			vadduws	v2.03			318	Vector Add Unsigned Word Saturate VX-form
000100 .... 01100 000000	I	..XX			vaddsbs	v2.03			313	Vector Add Signed Byte Saturate VX-form
000100 .... 01101 000000	I	..XX			vaddshs	v2.03			314	Vector Add Signed Halfword Saturate VX-form
000100 .... 01110 000000	I	..XX			vaddsws	v2.03			314	Vector Add Signed Word Saturate VX-form
000100 .... 10000 000000	I	..XX			vsububm	v2.03			323	Vector Subtract Unsigned Byte Modulo VX-form
000100 .... 10001 000000	I	..XX			vsubuhm	v2.03			323	Vector Subtract Unsigned Halfword Modulo VX-form
000100 .... 10010 000000	I	..XX			vsubuwm	v2.03			324	Vector Subtract Unsigned Word Modulo VX-form
000100 .... 10011 000000	I	..XX			vsubudm	v2.07			324	Vector Subtract Unsigned Doubleword Modulo VX-form
000100 .... 10100 000000	I	..XX			vsubuqm	v2.07			327	Vector Subtract Unsigned Quadword Modulo VX-form
000100 .... 10101 000000	I	..XX			vsubcuq	v2.07			328	Vector Subtract & write Carry-out Unsigned Quadword VX-form
000100 .... 10110 000000	I	..XX			vsubcuw	v2.03			321	Vector Subtract & Write Carry-out Unsigned Word VX-form
000100 .... 11000 000000	I	..XX			vsububs	v2.03			325	Vector Subtract Unsigned Byte Saturate VX-form
000100 .... 11001 000000	I	..XX			vsubuhs	v2.03			325	Vector Subtract Unsigned Halfword Saturate VX-form
000100 .... 11010 000000	I	..XX			vsubuws	v2.03			326	Vector Subtract Unsigned Word Saturate VX-form
000100 .... 11100 000000	I	..XX			vsubsbs	v2.03			321	Vector Subtract Signed Byte Saturate VX-form
000100 .... 11101 000000	I	..XX			vsubshs	v2.03			322	Vector Subtract Signed Halfword Saturate VX-form
000100 .... 11110 000000	I	..XX			vsubsws	v2.03			322	Vector Subtract Signed Word Saturate VX-form
000100 .... ///// 00000 000001	I	..XX			vmul10cuq	v3.0			474	Vector Multiply-by-10 & write Carry-out Unsigned Quadword VX-form
000100 .... 00001 000001	I	..XX			vmul10ecuq	v3.0			475	Vector Multiply-by-10 Extended & write Carry-out Unsigned Quadword VX-form
000100 .../// 00100 000001	I	..XX			vcmpuq	v3.1			391	Vector Compare Unsigned Quadword VX-form
000100 .../// 00101 000001	I	..XX			vcmpsq	v3.1			391	Vector Compare Signed Quadword VX-form
000100 .... ///// 01000 000001	I	..XX			vmul10uq	v3.0			474	Vector Multiply-by-10 Unsigned Quadword VX-form
000100 .... 01001 000001	I	..XX			vmul10euq	v3.0			475	Vector Multiply-by-10 Extended Unsigned Quadword VX-form
000100 .... 01101 000001	I	..XX			bcdcpsgn.	v3.0			476	Decimal Copy Sign VX-form

Table E.1: Power ISA Instruction Set Sorted by Opcode (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
000100 .... 1.000 000001	I	..XX			bcdadd.	v2.07			466	Decimal Add Modulo VX-form
000100 .... 1.001 000001	I	..XX			bcdsub.	v2.07			466	Decimal Subtract Modulo VX-form
000100 .... 1/010 000001	I	..XX			bcdus.	v3.0			479	Decimal Unsigned Shift VX-form
000100 .... 1.011 000001	I	..XX			bcds.	v3.0			478	Decimal Shift VX-form
000100 .... 1.100 000001	I	..XX			bcdtrunc.	v3.0			481	Decimal Truncate VX-form
000100 .... 1/101 000001	I	..XX			bcdutrunc.	v3.0			482	Decimal Unsigned Truncate VX-form
000100 ..... 00000 ..... 1/110 000001	I	..XX			bcdctsq.	v3.0			473	Decimal Convert To Signed Quadword VX-form
000100 ..... 00010 ..... 1.110 000001	I	..XX			bcdcfsq.	v3.0			472	Decimal Convert From Signed Quadword VX-form
000100 ..... 00100 ..... 1.110 000001	I	..XX			bcdctz.	v3.0			471	Decimal Convert To Zoned VX-form
000100 ..... 00101 ..... 1/110 000001	I	..XX			bcdctn.	v3.0			470	Decimal Convert To National VX-form
000100 ..... 00110 ..... 1.110 000001	I	..XX			bcdcfz.	v3.0			469	Decimal Convert From Zoned VX-form
000100 ..... 00111 ..... 1.110 000001	I	..XX			bcdcfn.	v3.0			468	Decimal Convert From National VX-form
000100 ..... 11111 ..... 1.110 000001	I	..XX			bcdsetsgn.	v3.0			477	Decimal Set Sign VX-form
000100 .... 1.111 000001	I	..XX			bcdsr.	v3.0			480	Decimal Shift & Round VX-form
000100 .... 00000 000010	I	..XX			vmaxub	v2.03			370	Vector Maximum Unsigned Byte VX-form
000100 .... 00001 000010	I	..XX			vmaxuh	v2.03			371	Vector Maximum Unsigned Halfword VX-form
000100 .... 00010 000010	I	..XX			vmaxuw	v2.03			372	Vector Maximum Unsigned Word VX-form
000100 .... 00011 000010	I	..XX			vmaxud	v2.07			373	Vector Maximum Unsigned Doubleword VX-form
000100 .... 00100 000010	I	..XX			vmaxsb	v2.03			370	Vector Maximum Signed Byte VX-form
000100 .... 00101 000010	I	..XX			vmaxsh	v2.03			371	Vector Maximum Signed Halfword VX-form
000100 .... 00110 000010	I	..XX			vmaxsw	v2.03			372	Vector Maximum Signed Word VX-form
000100 .... 00111 000010	I	..XX			vmaxsd	v2.07			373	Vector Maximum Signed Doubleword VX-form
000100 .... 01000 000010	I	..XX			vminub	v2.03			374	Vector Minimum Unsigned Byte VX-form
000100 .... 01001 000010	I	..XX			vminuh	v2.03			375	Vector Minimum Unsigned Halfword VX-form
000100 .... 01010 000010	I	..XX			vminuw	v2.03			376	Vector Minimum Unsigned Word VX-form
000100 .... 01011 000010	I	..XX			vminud	v2.07			377	Vector Minimum Unsigned Doubleword VX-form
000100 .... 01100 000010	I	..XX			vminsb	v2.03			374	Vector Minimum Signed Byte VX-form
000100 .... 01101 000010	I	..XX			vminsh	v2.03			375	Vector Minimum Signed Halfword VX-form
000100 .... 01110 000010	I	..XX			vminsw	v2.03			376	Vector Minimum Signed Word VX-form
000100 .... 01111 000010	I	..XX			vminsd	v2.07			377	Vector Minimum Signed Doubleword VX-form
000100 .... 10000 000010	I	..XX			vavgub	v2.03			365	Vector Average Unsigned Byte VX-form
000100 .... 10001 000010	I	..XX			vavguh	v2.03			366	Vector Average Unsigned Halfword VX-form
000100 .... 10010 000010	I	..XX			vavguw	v2.03			367	Vector Average Unsigned Word VX-form
000100 .... 10100 000010	I	..XX			vavgsb	v2.03			365	Vector Average Signed Byte VX-form
000100 .... 10101 000010	I	..XX			vavgsh	v2.03			366	Vector Average Signed Halfword VX-form
000100 .... 10110 000010	I	..XX			vavgsw	v2.03			367	Vector Average Signed Word VX-form
000100 ..... 00000 ..... 11000 000010	I	..XX			vclzlsbb	v3.0			441	Vector Count Leading Zero Least-Significant Bits Byte VX-form
000100 ..... 00001 ..... 11000 000010	I	..XX			vctzlsbb	v3.0			441	Vector Count Trailing Zero Least-Significant Bits Byte VX-form
000100 ..... 00110 ..... 11000 000010	I	..XX			vnegw	v3.0			361	Vector Negate Word VX-form
000100 ..... 00111 ..... 11000 000010	I	..XX			vnegd	v3.0			361	Vector Negate Doubleword VX-form
000100 ..... 01000 ..... 11000 000010	I	..XX			vprtybw	v3.0			447	Vector Parity Byte Word VX-form
000100 ..... 01001 ..... 11000 000010	I	..XX			vprtybd	v3.0			447	Vector Parity Byte Doubleword VX-form
000100 ..... 01010 ..... 11000 000010	I	..XX			vprtybq	v3.0			448	Vector Parity Byte Quadword VX-form
000100 ..... 10000 ..... 11000 000010	I	..XX			vextsb2w	v3.0			362	Vector Extend Sign Byte To Word VX-form
000100 ..... 10001 ..... 11000 000010	I	..XX			vextsh2w	v3.0			362	Vector Extend Sign Halfword To Word VX-form
000100 ..... 11000 ..... 11000 000010	I	..XX			vextsb2d	v3.0			363	Vector Extend Sign Byte To Doubleword VX-form

Table E.1: Power ISA Instruction Set Sorted by Opcode (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
000100 ..... 11001 ..... 11000 000010	I	..XX			vextsh2d	v3.0			363	Vector Extend Sign Halfword To Doubleword VX-form
000100 ..... 11010 ..... 11000 000010	I	..XX			vextsw2d	v3.0			364	Vector Extend Sign Word To Doubleword VX-form
000100 ..... 11011 ..... 11000 000010	I	..XX			vextsd2q	v3.1			364	Vector Extend Sign Doubleword to Quadword VX-form
000100 ..... 11100 ..... 11000 000010	I	..XX			vctzb	v3.0			438	Vector Count Trailing Zeros Byte VX-form
000100 ..... 11101 ..... 11000 000010	I	..XX			vctzh	v3.0			438	Vector Count Trailing Zeros Halfword VX-form
000100 ..... 11110 ..... 11000 000010	I	..XX			vctzw	v3.0			439	Vector Count Trailing Zeros Word VX-form
000100 ..... 11111 ..... 11000 000010	I	..XX			vctzd	v3.0			440	Vector Count Trailing Zeros Doubleword VX-form
000100 ..... 00000 ..... 11001 000010	I	..XX			vexpandbm	v3.1			454	Vector Expand Byte Mask VX-form
000100 ..... 00001 ..... 11001 000010	I	..XX			vexpandhm	v3.1			454	Vector Expand Halfword Mask VX-form
000100 ..... 00010 ..... 11001 000010	I	..XX			vexpandwm	v3.1			455	Vector Expand Word Mask VX-form
000100 ..... 00011 ..... 11001 000010	I	..XX			vexpanddm	v3.1			455	Vector Expand Doubleword Mask VX-form
000100 ..... 00100 ..... 11001 000010	I	..XX			vexpandqm	v3.1			456	Vector Expand Quadword Mask VX-form
000100 ..... 01000 ..... 11001 000010	I	..XX			vextractbm	v3.1			459	Vector Extract Byte Mask VX-form
000100 ..... 01001 ..... 11001 000010	I	..XX			vextracthm	v3.1			459	Vector Extract Halfword Mask VX-form
000100 ..... 01010 ..... 11001 000010	I	..XX			vextractwm	v3.1			460	Vector Extract Word Mask VX-form
000100 ..... 01011 ..... 11001 000010	I	..XX			vextractdm	v3.1			460	Vector Extract Doubleword Mask VX-form
000100 ..... 01100 ..... 11001 000010	I	..XX			vextractqm	v3.1			461	Vector Extract Quadword Mask VX-form
000100 ..... 10000 ..... 11001 000010	I	..XX			mtvsrbm	v3.1			451	Move to VSR Byte Mask VX-form
000100 ..... 10001 ..... 11001 000010	I	..XX			mtvsrhm	v3.1			451	Move to VSR Halfword Mask VX-form
000100 ..... 10010 ..... 11001 000010	I	..XX			mtvsrwm	v3.1			452	Move to VSR Word Mask VX-form
000100 ..... 10011 ..... 11001 000010	I	..XX			mtvsrdm	v3.1			452	Move to VSR Doubleword Mask VX-form
000100 ..... 10100 ..... 11001 000010	I	..XX			mtvsrqm	v3.1			453	Move to VSR Quadword Mask VX-form
000100 ..... 1100. ..... 11001 000010	I	..XX			vcntmbb	v3.1			457	Vector Count Mask Bits Byte VX-form
000100 ..... 1101. ..... 11001 000010	I	..XX			vcntmbh	v3.1			457	Vector Count Mask Bits Halfword VX-form
000100 ..... 1110. ..... 11001 000010	I	..XX			vcntmbw	v3.1			458	Vector Count Mask Bits Word VX-form
000100 ..... 1111. ..... 11001 000010	I	..XX			vcntmbd	v3.1			458	Vector Count Mask Bits Doubleword VX-form
000100 .... 11010 000010	I	..XX			vshasigmaw	v2.07			428	Vector SHA-256 Sigma Word VX-form
000100 .... 11011 000010	I	..XX			vshasigmad	v2.07			427	Vector SHA-512 Sigma Doubleword VX-form
000100 ..... ///// 11100 000010	I	..XX			vclzb	v2.07			435	Vector Count Leading Zeros Byte VX-form
000100 ..... ///// 11101 000010	I	..XX			vclzh	v2.07			435	Vector Count Leading Zeros Halfword VX-form
000100 ..... ///// 11110 000010	I	..XX			vclzw	v2.07			436	Vector Count Leading Zeros Word VX-form
000100 ..... ///// 11111 000010	I	..XX			vclzd	v2.07			437	Vector Count Leading Zeros Doubleword VX-form
000100 .... 10000 000011	I	..XX			vabsdub	v3.0			368	Vector Absolute Difference Unsigned Byte VX-form
000100 .... 10001 000011	I	..XX			vabsduh	v3.0			368	Vector Absolute Difference Unsigned Halfword VX-form
000100 .... 10010 000011	I	..XX			vabsduw	v3.0			369	Vector Absolute Difference Unsigned Word VX-form
000100 ..... ///// 11100 000011	I	..XX			vpopcntb	v2.07			445	Vector Population Count Byte VX-form
000100 ..... ///// 11101 000011	I	..XX			vpopcnth	v2.07			445	Vector Population Count Halfword VX-form
000100 ..... ///// 11110 000011	I	..XX			vpopcntw	v2.07			446	Vector Population Count Word VX-form
000100 ..... ///// 11111 000011	I	..XX			vpopcntd	v2.07			446	Vector Population Count Doubleword VX-form
000100 .... 00000 000100	I	..XX			vrlb	v2.03			395	Vector Rotate Left Byte VX-form
000100 .... 00001 000100	I	..XX			vrlh	v2.03			395	Vector Rotate Left Halfword VX-form
000100 .... 00010 000100	I	..XX			vrlw	v2.03			396	Vector Rotate Left Word VX-form
000100 .... 00011 000100	I	..XX			vrld	v2.07			396	Vector Rotate Left Doubleword VX-form
000100 .... 00100 000100	I	..XX			vslb	v2.03			402	Vector Shift Left Byte VX-form
000100 .... 00101 000100	I	..XX			vslh	v2.03			402	Vector Shift Left Halfword VX-form
000100 .... 00110 000100	I	..XX			vslw	v2.03			403	Vector Shift Left Word VX-form
000100 .... 00111 000100	I	..XX			vsl	v2.03			290	Vector Shift Left VX-form
000100 .... 01000 000100	I	..XX			vsrb	v2.03			405	Vector Shift Right Byte VX-form

Table E.1: Power ISA Instruction Set Sorted by Opcode (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
000100 .... 01001 000100	I	..XX			vsrh	v2.03			405	Vector Shift Right Halfword VX-form
000100 .... 01010 000100	I	..XX			vsrw	v2.03			406	Vector Shift Right Word VX-form
000100 .... 01011 000100	I	..XX			vsr	v2.03			290	Vector Shift Right VX-form
000100 .... 01100 000100	I	..XX			vsrab	v2.03			408	Vector Shift Right Algebraic Byte VX-form
000100 .... 01101 000100	I	..XX			vsrah	v2.03			408	Vector Shift Right Algebraic Halfword VX-form
000100 .... 01110 000100	I	..XX			vsraw	v2.03			409	Vector Shift Right Algebraic Word VX-form
000100 .... 01111 000100	I	..XX			vsrad	v2.07			409	Vector Shift Right Algebraic Doubleword VX-form
000100 .... 10000 000100	I	..XX			vand	v2.03			392	Vector Logical AND VX-form
000100 .... 10001 000100	I	..XX			vandc	v2.03			392	Vector Logical AND with Complement VX-form
000100 .... 10010 000100	I	..XX			vor	v2.03			393	Vector Logical OR VX-form
000100 .... 10011 000100	I	..XX			vxor	v2.03			394	Vector Logical XOR VX-form
000100 .... 10100 000100	I	..XX			vnor	v2.03			394	Vector Logical NOR VX-form
000100 .... 10101 000100	I	..XX			vorc	v2.07			393	Vector Logical OR with Complement VX-form
000100 .... 10110 000100	I	..XX			vnand	v2.07			393	Vector Logical NAND VX-form
000100 .... 10111 000100	I	..XX			vsld	v2.07			403	Vector Shift Left Doubleword VX-form
000100 ..... ///// ///// 11000 000100	I	..XX			mfvscr	v2.03			483	Move From Vector Status and Control Register VX-form
000100 ///// ///// 11001 000100	I	..XX			mtvscr	v2.03			483	Move To Vector Status and Control Register VX-form
000100 .... 11010 000100	I	..XX			veqv	v2.07			393	Vector Logical Equivalence VX-form
000100 .... 11011 000100	I	..XX			vsrd	v2.07			406	Vector Shift Right Doubleword VX-form
000100 .... 11100 000100	I	..XX			vsrv	v3.0			292	Vector Shift Right Variable VX-form
000100 .... 11101 000100	I	..XX			vslv	v3.0			292	Vector Shift Left Variable VX-form
000100 .... 11110 000100	I	..XX			vclzdm	v3.1			437	Vector Count Leading Zeros Doubleword under bit Mask VX-form
000100 .... 11111 000100	I	..XX			vctzdm	v3.1			440	Vector Count Trailing Zeros Doubleword under bit Mask VX-form
000100 .... 00000 000101	I	..XX			vrlq	v3.1			397	Vector Rotate Left Quadword VX-form
000100 .... 00001 000101	I	..XX			vrlqmi	v3.1			401	Vector Rotate Left Quadword then Mask Insert VX-form
000100 .... 00010 000101	I	..XX			vrlwmi	v3.0			400	Vector Rotate Left Word then Mask Insert VX-form
000100 .... 00011 000101	I	..XX			vrldmi	v3.0			401	Vector Rotate Left Doubleword then Mask Insert VX-form
000100 .... 00100 000101	I	..XX			vslq	v3.1			404	Vector Shift Left Quadword VX-form
000100 .... 00101 000101	I	..XX			vrlqnm	v3.1			399	Vector Rotate Left Quadword then AND with Mask VX-form
000100 .... 00110 000101	I	..XX			vrlwnm	v3.0			398	Vector Rotate Left Word then AND with Mask VX-form
000100 .... 00111 000101	I	..XX			vrldnm	v3.0			398	Vector Rotate Left Doubleword then AND with Mask VX-form
000100 .... 01000 000101	I	..XX			vsrq	v3.1			407	Vector Shift Right Quadword VX-form
000100 .... 01100 000101	I	..XX			vsraq	v3.1			410	Vector Shift Right Algebraic Quadword VX-form
000100 .... .0000 000110	I	..XX			vcmpequb[.]	v2.03			378	Vector Compare Equal Unsigned Byte VC-form
000100 .... .0001 000110	I	..XX			vcmpequh[.]	v2.03			379	Vector Compare Equal Unsigned Halfword VC-form
000100 .... .0010 000110	I	..XX			vcmpequw[.]	v2.03			380	Vector Compare Equal Unsigned Word VC-form
000100 .... .0011 000110	I	..XX			vcmpeqfp[.]	v2.03			419	Vector Compare Equal Floating-Point VC-form
000100 .... .0111 000110	I	..XX			vcmpgefp[.]	v2.03			419	Vector Compare Greater Than or Equal Floating-Point VC-form
000100 .... .1000 000110	I	..XX			vcmpgtub[.]	v2.03			383	Vector Compare Greater Than Unsigned Byte VC-form
000100 .... .1001 000110	I	..XX			vcmpgtuh[.]	v2.03			384	Vector Compare Greater Than Unsigned Halfword VC-form
000100 .... .1010 000110	I	..XX			vcmpgtuw[.]	v2.03			385	Vector Compare Greater Than Unsigned Word VC-form
000100 .... .1011 000110	I	..XX			vcmpgtfp[.]	v2.03			420	Vector Compare Greater Than Floating-Point VC-form

Table E.1: Power ISA Instruction Set Sorted by Opcode (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
000100 .... .1100 000110	I	..XX			vcmpgtsb[.]	v2.03			383	Vector Compare Greater Than Signed Byte VC-form
000100 .... .1101 000110	I	..XX			vcmpgtsh[.]	v2.03			384	Vector Compare Greater Than Signed Halfword VC-form
000100 .... .1110 000110	I	..XX			vcmpgtsw[.]	v2.03			385	Vector Compare Greater Than Signed Word VC-form
000100 .... .1111 000110	I	..XX			vcmpbfp[.]	v2.03			418	Vector Compare Bounds Floating-Point VC-form
000100 .... .0000 000111	I	..XX			vcmpneb[.]	v3.0			388	Vector Compare Not Equal Byte VC-form
000100 .... .0001 000111	I	..XX			vcmpneh[.]	v3.0			389	Vector Compare Not Equal Halfword VC-form
000100 .... .0010 000111	I	..XX			vcmpnew[.]	v3.0			390	Vector Compare Not Equal Word VC-form
000100 .... .0011 000111	I	..XX			vcmpequd[.]	v2.07			381	Vector Compare Equal Unsigned Doubleword VC-form
000100 .... .0100 000111	I	..XX			vcmpnezb[.]	v3.0			388	Vector Compare Not Equal or Zero Byte VC-form
000100 .... .0101 000111	I	..XX			vcmpnezh[.]	v3.0			389	Vector Compare Not Equal or Zero Halfword VC-form
000100 .... .0110 000111	I	..XX			vcmpnezw[.]	v3.0			390	Vector Compare Not Equal or Zero Word VC-form
000100 .... .0111 000111	I	..XX			vcmpequq[.]	v3.1			382	Vector Compare Equal Quadword VC-form
000100 .... .1010 000111	I	..XX			vcmpgtuq[.]	v3.1			387	Vector Compare Greater Than Unsigned Quadword VC-form
000100 .... .1011 000111	I	..XX			vcmpgtud[.]	v2.07			386	Vector Compare Greater Than Unsigned Doubleword VC-form
000100 .... .1110 000111	I	..XX			vcmpgtsq[.]	v3.1			387	Vector Compare Greater Than Signed Quadword VC-form
000100 .... .1111 000111	I	..XX			vcmpgtsd[.]	v2.07			386	Vector Compare Greater Than Signed Doubleword VC-form
000100 .... 00000 001000	I	..XX			vmuloub	v2.03			330	Vector Multiply Odd Unsigned Byte VX-form
000100 .... 00001 001000	I	..XX			vmulouh	v2.03			332	Vector Multiply Odd Unsigned Halfword VX-form
000100 .... 00010 001000	I	..XX			vmulouw	v2.07			334	Vector Multiply Odd Unsigned Word VX-form
000100 .... 00011 001000	I	..XX			vmuloud	v3.1			335	Vector Multiply Odd Unsigned Doubleword VX-form
000100 .... 00100 001000	I	..XX			vmulosb	v2.03			329	Vector Multiply Odd Signed Byte VX-form
000100 .... 00101 001000	I	..XX			vmulosh	v2.03			331	Vector Multiply Odd Signed Halfword VX-form
000100 .... 00110 001000	I	..XX			vmulosw	v2.07			333	Vector Multiply Odd Signed Word VX-form
000100 .... 00111 001000	I	..XX			vmulosd	v3.1			336	Vector Multiply Odd Signed Doubleword VX-form
000100 .... 01000 001000	I	..XX			vmuleub	v2.03			330	Vector Multiply Even Unsigned Byte VX-form
000100 .... 01001 001000	I	..XX			vmuleuh	v2.03			332	Vector Multiply Even Unsigned Halfword VX-form
000100 .... 01010 001000	I	..XX			vmuleuw	v2.07			334	Vector Multiply Even Unsigned Word VX-form
000100 .... 01011 001000	I	..XX			vmuleud	v3.1			335	Vector Multiply Even Unsigned Doubleword VX-form
000100 .... 01100 001000	I	..XX			vmulesb	v2.03			329	Vector Multiply Even Signed Byte VX-form
000100 .... 01101 001000	I	..XX			vmulesh	v2.03			331	Vector Multiply Even Signed Halfword VX-form
000100 .... 01110 001000	I	..XX			vmulesw	v2.07			333	Vector Multiply Even Signed Word VX-form
000100 .... 01111 001000	I	..XX			vmulesd	v3.1			336	Vector Multiply Even Signed Doubleword VX-form
000100 .... 10000 001000	I	..XX			vpmsumb	v2.07			430	Vector Polynomial Multiply-Sum Byte VX-form
000100 .... 10001 001000	I	..XX			vpmsumh	v2.07			430	Vector Polynomial Multiply-Sum Halfword VX-form
000100 .... 10010 001000	I	..XX			vpmsumw	v2.07			431	Vector Polynomial Multiply-Sum Word VX-form
000100 .... 10011 001000	I	..XX			vpmsumd	v2.07			431	Vector Polynomial Multiply-Sum Doubleword VX-form
000100 .... 10100 001000	I	..XX			vcipher	v2.07			424	Vector AES Cipher VX-form
000100 .... 10101 001000	I	..XX			vncipher	v2.07			425	Vector AES Inverse Cipher VX-form
000100 .... ///// 10111 001000	I	..XX			vsbox	v2.07			426	Vector AES SubBytes VX-form
000100 .... 11000 001000	I	..XX			vsum4ubs	v2.03			360	Vector Sum across Quarter Unsigned Byte Saturate VX-form
000100 .... 11001 001000	I	..XX			vsum4shs	v2.03			359	Vector Sum across Quarter Signed Halfword Saturate VX-form
000100 .... 11010 001000	I	..XX			vsum2sws	v2.03			358	Vector Sum across Half Signed Word Saturate VX-form
000100 .... 11100 001000	I	..XX			vsum4sbs	v2.03			359	Vector Sum across Quarter Signed Byte Saturate VX-form

Table E.1: Power ISA Instruction Set Sorted by Opcode (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
000100 .... 11110 001000	I	..XX			vsumsws	v2.03			357	Vector Sum across Signed Word Saturate VX-form
000100 .... 00010 001001	I	..XX			vmuluwm	v2.07			337	Vector Multiply Unsigned Word Modulo VX-form
000100 .... 00111 001001	I	..XX			vmulld	v3.1			340	Vector Multiply Low Doubleword VX-form
000100 .... 01010 001001	I	..XX			vmulhuw	v3.1			338	Vector Multiply High Unsigned Word VX-form
000100 .... 01011 001001	I	..XX			vmulhud	v3.1			339	Vector Multiply High Unsigned Doubleword VX-form
000100 .... 01110 001001	I	..XX			vmulhsw	v3.1			338	Vector Multiply High Signed Word VX-form
000100 .... 01111 001001	I	..XX			vmulhsd	v3.1			339	Vector Multiply High Signed Doubleword VX-form
000100 .... 10100 001001	I	..XX			vcipherlast	v2.07			424	Vector AES Cipher Last VX-form
000100 .... 10101 001001	I	..XX			vncipherlast	v2.07			425	Vector AES Inverse Cipher Last VX-form
000100 .... 00000 001010	I	..XX			vaddfp	v2.03			411	Vector Add Floating-Point VX-form
000100 .... 00001 001010	I	..XX			vsubfp	v2.03			411	Vector Subtract Floating-Point VX-form
000100 ..... ///// 00100 001010	I	..XX			vrefp	v2.03			423	Vector Reciprocal Estimate Floating-Point VX-form
000100 ..... ///// 00101 001010	I	..XX			vrsqrtefp	v2.03			423	Vector Reciprocal Square Root Estimate Floating-Point VX-form
000100 ..... ///// 00110 001010	I	..XX			vexptefp	v2.03			421	Vector 2 Raised to the Exponent Estimate Floating-Point VX-form
000100 ..... ///// 00111 001010	I	..XX			vlogefp	v2.03			422	Vector Log Base 2 Estimate Floating-Point VX-form
000100 ..... ///// 01000 001010	I	..XX			vrfin	v2.03			416	Vector Round to Floating-Point Integer Nearest VX-form
000100 ..... ///// 01001 001010	I	..XX			vrfiz	v2.03			417	Vector Round to Floating-Point Integer toward Zero VX-form
000100 ..... ///// 01010 001010	I	..XX			vrfip	v2.03			417	Vector Round to Floating-Point Integer toward +Infinity VX-form
000100 ..... ///// 01011 001010	I	..XX			vrfim	v2.03			416	Vector Round to Floating-Point Integer toward -Infinity VX-form
000100 .... 01100 001010	I	..XX			vcfux	v2.03			415	Vector Convert with round to nearest From Unsigned Word to floating-point format VX-form
000100 .... 01101 001010	I	..XX			vcfsx	v2.03			415	Vector Convert with round to nearest From Signed Word to floating-point format VX-form
000100 .... 01110 001010	I	..XX			vctuxs	v2.03			414	Vector Convert with round to zero from floating-point To Unsigned Word format Saturate VX-form
000100 .... 01111 001010	I	..XX			vctsxs	v2.03			414	Vector Convert with round to zero from floating-point To Signed Word format Saturate VX-form
000100 .... 10000 001010	I	..XX			vmaxfp	v2.03			413	Vector Maximum Floating-Point VX-form
000100 .... 10001 001010	I	..XX			vminfp	v2.03			413	Vector Minimum Floating-Point VX-form
000100 .... 00000 001011	I	..XX			vdivuq	v3.1			352	Vector Divide Unsigned Quadword VX-form
000100 .... 00010 001011	I	..XX			vdivuw	v3.1			348	Vector Divide Unsigned Word VX-form
000100 .... 00011 001011	I	..XX			vdivud	v3.1			350	Vector Divide Unsigned Doubleword VX-form
000100 .... 00100 001011	I	..XX			vdivsq	v3.1			352	Vector Divide Signed Quadword VX-form
000100 .... 00110 001011	I	..XX			vdivsw	v3.1			348	Vector Divide Signed Word VX-form
000100 .... 00111 001011	I	..XX			vdivsd	v3.1			350	Vector Divide Signed Doubleword VX-form
000100 .... 01000 001011	I	..XX			vdiveuq	v3.1			353	Vector Divide Extended Unsigned Quadword VX-form
000100 .... 01010 001011	I	..XX			vdiveuw	v3.1			349	Vector Divide Extended Unsigned Word VX-form
000100 .... 01011 001011	I	..XX			vdiveud	v3.1			351	Vector Divide Extended Unsigned Doubleword VX-form
000100 .... 01100 001011	I	..XX			vdivesq	v3.1			353	Vector Divide Extended Signed Quadword VX-form
000100 .... 01110 001011	I	..XX			vdivesw	v3.1			349	Vector Divide Extended Signed Word VX-form
000100 .... 01111 001011	I	..XX			vdivesd	v3.1			351	Vector Divide Extended Signed Doubleword VX-form
000100 .... 11000 001011	I	..XX			vmoduq	v3.1			356	Vector Modulo Unsigned Quadword VX-form
000100 .... 11010 001011	I	..XX			vmoduw	v3.1			354	Vector Modulo Unsigned Word VX-form
000100 .... 11011 001011	I	..XX			vmodud	v3.1			355	Vector Modulo Unsigned Doubleword VX-form

Table E.1: Power ISA Instruction Set Sorted by Opcode (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
000100 .... 11100 001011	I	..XX			vmodsq	v3.1			356	Vector Modulo Signed Quadword VX-form
000100 .... 11110 001011	I	..XX			vmodsw	v3.1			354	Vector Modulo Signed Word VX-form
000100 .... 11111 001011	I	..XX			vmodsd	v3.1			355	Vector Modulo Signed Doubleword VX-form
000100 .... 00000 001100	I	..XX			vmrghb	v2.03			279	Vector Merge High Byte VX-form
000100 .... 00001 001100	I	..XX			vmrghh	v2.03			280	Vector Merge High Halfword VX-form
000100 .... 00010 001100	I	..XX			vmrghw	v2.03			281	Vector Merge High Word VX-form
000100 .... 00100 001100	I	..XX			vmrglb	v2.03			279	Vector Merge Low Byte VX-form
000100 .... 00101 001100	I	..XX			vmrglh	v2.03			280	Vector Merge Low Halfword VX-form
000100 .... 00110 001100	I	..XX			vmrglw	v2.03			281	Vector Merge Low Word VX-form
000100 ..... /.... ..... 01000 001100	I	..XX			vspltb	v2.03			283	Vector Splat Byte VX-form
000100 ..... //... ..... 01001 001100	I	..XX			vsplth	v2.03			283	Vector Splat Halfword VX-form
000100 ..... ///.. ..... 01010 001100	I	..XX			vspltw	v2.03			284	Vector Splat Word VX-form
000100 .... ///// 01100 001100	I	..XX			vspltisb	v2.03			285	Vector Splat Immediate Signed Byte VX-form
000100 .... ///// 01101 001100	I	..XX			vspltish	v2.03			285	Vector Splat Immediate Signed Halfword VX-form
000100 .... ///// 01110 001100	I	..XX			vspltisw	v2.03			285	Vector Splat Immediate Signed Word VX-form
000100 .... 10000 001100	I	..XX			vslo	v2.03			291	Vector Shift Left by Octet VX-form
000100 .... 10001 001100	I	..XX			vsro	v2.03			291	Vector Shift Right by Octet VX-form
000100 ..... //... ..... 10011 001100	I	..XX			vgnb	v3.1			434	Vector Gather every Nth Bit VX-form
000100 ..... ///// 10100 001100	I	..XX			vgbbd	v2.07			433	Vector Gather Bits by Bytes by Doubleword VX-form
000100 .... 10101 001100	I	..XX			vbpermq	v2.07			450	Vector Bit Permute Quadword VX-form
000100 .... 10111 001100	I	..XX			vbpermd	v3.0			449	Vector Bit Permute Doubleword VX-form
000100 .... 11010 001100	I	..XX			vmrgow	v2.07			282	Vector Merge Odd Word VX-form
000100 .... 11110 001100	I	..XX			vmrgew	v2.07			282	Vector Merge Even Word VX-form
000100 ..... 00000 ..... .0000 001101	I	..XX			vstribl[.]	v3.1			462	Vector String Isolate Byte Left-justified VC-form
000100 ..... 00001 ..... .0000 001101	I	..XX			vstribr[.]	v3.1			462	Vector String Isolate Byte Right-justified VC-form
000100 ..... 00010 ..... .0000 001101	I	..XX			vstrihl[.]	v3.1			463	Vector String Isolate Halfword Left-justified VC-form
000100 ..... 00011 ..... .0000 001101	I	..XX			vstrihr[.]	v3.1			463	Vector String Isolate Halfword Right-justified VC-form
000100 .... 00110 001101	I	..XX			vclrlb	v3.1			464	Vector Clear Leftmost Bytes VX-form
000100 .... 00111 001101	I	..XX			vclrrb	v3.1			464	Vector Clear Rightmost Bytes VX-form
000100 ..... /.... ..... 01000 001101	I	..XX			vextractub	v3.0			294	Vector Extract Unsigned Byte to VSR using immediate-specified index VX-form
000100 ..... /.... ..... 01001 001101	I	..XX			vextractuh	v3.0			294	Vector Extract Unsigned Halfword to VSR using immediate-specified index VX-form
000100 ..... /.... ..... 01010 001101	I	..XX			vextractuw	v3.0			295	Vector Extract Unsigned Word to VSR using immediate-specified index VX-form
000100 ..... /.... ..... 01011 001101	I	..XX			vextractd	v3.0			295	Vector Extract Doubleword to VSR using immediate-specified index VX-form
000100 ..... /.... ..... 01100 001101	I	..XX			vinsertb	v3.0			303	Vector Insert Byte from VSR using immediate-specified index VX-form
000100 ..... /.... ..... 01101 001101	I	..XX			vinserth	v3.0			303	Vector Insert Halfword from VSR using immediate-specified index VX-form
000100 ..... /.... ..... 01110 001101	I	..XX			vinsertw	v3.0			304	Vector Insert Word from VSR using immediate-specified index VX-form
000100 ..... /.... ..... 01111 001101	I	..XX			vinsertd	v3.0			304	Vector Insert Doubleword from VSR using immediate-specified index VX-form
000100 .... 10101 001101	I	..XX			vcfuged	v3.1			444	Vector Centrifuge Doubleword VX-form
000100 .... 10110 001101	I	..XX			vpextd	v3.1			443	Vector Parallel Bits Extract Doubleword VX-form
000100 .... 10111 001101	I	..XX			vpdepd	v3.1			442	Vector Parallel Bits Deposit Doubleword VX-form

Table E.1: Power ISA Instruction Set Sorted by Opcode (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
000100 .... 11000 001101	I	..XX			vextublx	v3.0			296	Vector Extract Unsigned Byte to GPR using GPR-specified Left-Index VX-form
000100 .... 11001 001101	I	..XX			vextuhlx	v3.0			297	Vector Extract Unsigned Halfword to GPR using GPR-specified Left-Index VX-form
000100 .... 11010 001101	I	..XX			vextuwlx	v3.0			298	Vector Extract Unsigned Word to GPR using GPR-specified Left-Index VX-form
000100 .... 11100 001101	I	..XX			vextubrx	v3.0			296	Vector Extract Unsigned Byte to GPR using GPR-specified Right-Index VX-form
000100 .... 11101 001101	I	..XX			vextuhrx	v3.0			297	Vector Extract Unsigned Halfword to GPR using GPR-specified Right-Index VX-form
000100 .... 11110 001101	I	..XX			vextuwrx	v3.0			298	Vector Extract Unsigned Word to GPR using GPR-specified Right-Index VX-form
000100 .... 00000 001110	I	..XX			vpkuhum	v2.03			272	Vector Pack Unsigned Halfword Unsigned Modulo VX-form
000100 .... 00001 001110	I	..XX			vpkuwum	v2.03			273	Vector Pack Unsigned Word Unsigned Modulo VX-form
000100 .... 00010 001110	I	..XX			vpkuhus	v2.03			272	Vector Pack Unsigned Halfword Unsigned Saturate VX-form
000100 .... 00011 001110	I	..XX			vpkuwus	v2.03			273	Vector Pack Unsigned Word Unsigned Saturate VX-form
000100 .... 00100 001110	I	..XX			vpkshus	v2.03			269	Vector Pack Signed Halfword Unsigned Saturate VX-form
000100 .... 00101 001110	I	..XX			vpkswus	v2.03			270	Vector Pack Signed Word Unsigned Saturate VX-form
000100 .... 00110 001110	I	..XX			vpkshss	v2.03			269	Vector Pack Signed Halfword Signed Saturate VX-form
000100 .... 00111 001110	I	..XX			vpkswss	v2.03			270	Vector Pack Signed Word Signed Saturate VX-form
000100 ..... ///// 01000 001110	I	..XX			vupkhsb	v2.03			275	Vector Unpack High Signed Byte VX-form
000100 ..... ///// 01001 001110	I	..XX			vupkhsh	v2.03			276	Vector Unpack High Signed Halfword VX-form
000100 ..... ///// 01010 001110	I	..XX			vupklsb	v2.03			275	Vector Unpack Low Signed Byte VX-form
000100 ..... ///// 01011 001110	I	..XX			vupklsh	v2.03			276	Vector Unpack Low Signed Halfword VX-form
000100 .... 01100 001110	I	..XX			vpkpx	v2.03			268	Vector Pack Pixel VX-form
000100 ..... ///// 01101 001110	I	..XX			vupkhpx	v2.03			278	Vector Unpack High Pixel VX-form
000100 ..... ///// 01111 001110	I	..XX			vupklpx	v2.03			278	Vector Unpack Low Pixel VX-form
000100 .... 10001 001110	I	..XX			vpkudum	v2.07			274	Vector Pack Unsigned Doubleword Unsigned Modulo VX-form
000100 .... 10011 001110	I	..XX			vpkudus	v2.07			274	Vector Pack Unsigned Doubleword Unsigned Saturate VX-form
000100 .... 10101 001110	I	..XX			vpksdus	v2.07			271	Vector Pack Signed Doubleword Unsigned Saturate VX-form
000100 .... 10111 001110	I	..XX			vpksdss	v2.07			271	Vector Pack Signed Doubleword Signed Saturate VX-form
000100 ..... ///// 11001 001110	I	..XX			vupkhsw	v2.07			277	Vector Unpack High Signed Word VX-form
000100 ..... ///// 11011 001110	I	..XX			vupklsw	v2.07			277	Vector Unpack Low Signed Word VX-form
000100 .... 00000 001111	I	..XX			vinsbvlx	v3.1			310	Vector Insert Byte from VSR using GPR-specified Left-Index VX-form
000100 .... 00001 001111	I	..XX			vinshvlx	v3.1			311	Vector Insert Halfword from VSR using GPR-specified Left-Index VX-form
000100 .... 00010 001111	I	..XX			vinswvlx	v3.1			312	Vector Insert Word from VSR using GPR-specified Left-Index VX-form
000100 ..... /.... ..... 00011 001111	I	..XX			vinsw	v3.1			309	Vector Insert Word from GPR using immediate-specified index VX-form
000100 .... 00100 001111	I	..XX			vinsbvrx	v3.1			310	Vector Insert Byte from VSR using GPR-specified Right-Index VX-form

Table E.1: Power ISA Instruction Set Sorted by Opcode (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
000100 .... 00101 001111	I	..XX			vinshvrx	v3.1			311	Vector Insert Halfword from VSR using GPR-specified Right-Index VX-form
000100 .... 00110 001111	I	..XX			vinswvrx	v3.1			312	Vector Insert Word from VSR using GPR-specified Right-Index VX-form
000100 ..... /.... ..... 00111 001111	I	..XX			vinsd	v3.1			309	Vector Insert Doubleword from GPR using immediate-specified index VX-form
000100 .... 01000 001111	I	..XX			vinsblx	v3.1			305	Vector Insert Byte from GPR using GPR-specified Left-Index VX-form
000100 .... 01001 001111	I	..XX			vinshlx	v3.1			306	Vector Insert Halfword from GPR using GPR-specified Left-Index VX-form
000100 .... 01010 001111	I	..XX			vinswlx	v3.1			307	Vector Insert Word from GPR using GPR-specified Left-Index VX-form
000100 .... 01011 001111	I	..XX			vinsdlx	v3.1			308	Vector Insert Doubleword from GPR using GPR-specified Left-Index VX-form
000100 .... 01100 001111	I	..XX			vinsbrx	v3.1			305	Vector Insert Byte from GPR using GPR-specified Right-Index VX-form
000100 .... 01101 001111	I	..XX			vinshrx	v3.1			306	Vector Insert Halfword from GPR using GPR-specified Right-Index VX-form
000100 .... 01110 001111	I	..XX			vinswrx	v3.1			307	Vector Insert Word from GPR using GPR-specified Right-Index VX-form
000100 .... 01111 001111	I	..XX			vinsdrx	v3.1			308	Vector Insert Doubleword from GPR using GPR-specified Right-Index VX-form
000100 .... 01010.	I	..XX			mtvsrbmi	v3.1			453	Move To VSR Byte Mask Immediate DX-form
000100 .... 00... 010110	I	..XX			vsldbi	v3.1			289	Vector Shift Left Double by Bit Immediate VN-form
000100 .... 01... 010110	I	..XX			vsrdbi	v3.1			289	Vector Shift Right Double by Bit Immediate VN-form
000100 010111	I	..XX			vmsumcud	v3.1			347	Vector Multiply-Sum & write Carry-out Unsigned Doubleword VA-form
000100 011000	I	..XX			vextdubvlx	v3.1			299	Vector Extract Double Unsigned Byte to VSR using GPR-specified Left-Index VA-form
000100 011001	I	..XX			vextdubvrx	v3.1			299	Vector Extract Double Unsigned Byte to VSR using GPR-specified Right-Index VA-form
000100 011010	I	..XX			vextduhvlx	v3.1			300	Vector Extract Double Unsigned Halfword to VSR using GPR-specified Left-Index VA-form
000100 011011	I	..XX			vextduhvrx	v3.1			300	Vector Extract Double Unsigned Halfword to VSR using GPR-specified Right-Index VA-form
000100 011100	I	..XX			vextduwvlx	v3.1			301	Vector Extract Double Unsigned Word to VSR using GPR-specified Left-Index VA-form
000100 011101	I	..XX			vextduwvrx	v3.1			301	Vector Extract Double Unsigned Word to VSR using GPR-specified Right-Index VA-form
000100 011110	I	..XX			vextddvlx	v3.1			302	Vector Extract Double Doubleword to VSR using GPR-specified Left-Index VA-form
000100 011111	I	..XX			vextddvrx	v3.1			302	Vector Extract Double Doubleword to VSR using GPR-specified Right-Index VA-form
000100 100000	I	..XX			vmhaddshs	v2.03			341	Vector Multiply-High-Add Signed Halfword Saturate VA-form
000100 100001	I	..XX			vmhraddshs	v2.03			341	Vector Multiply-High-Round-Add Signed Halfword Saturate VA-form
000100 100010	I	..XX			vmladduhm	v2.03			342	Vector Multiply-Low-Add Unsigned Halfword Modulo VA-form

Table E.1: Power ISA Instruction Set Sorted by Opcode (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
000100 100011	I	..XX			vmsumudm	v3.0B			346	Vector Multiply-Sum Unsigned Doubleword Modulo VA-form
000100 100100	I	..XX			vmsumubm	v2.03			342	Vector Multiply-Sum Unsigned Byte Modulo VA-form
000100 100101	I	..XX			vmsummbm	v2.03			343	Vector Multiply-Sum Mixed Byte Modulo VA-form
000100 100110	I	..XX			vmsumuhm	v2.03			344	Vector Multiply-Sum Unsigned Halfword Modulo VA-form
000100 100111	I	..XX			vmsumuhs	v2.03			345	Vector Multiply-Sum Unsigned Halfword Saturate VA-form
000100 101000	I	..XX			vmsumshm	v2.03			343	Vector Multiply-Sum Signed Halfword Modulo VA-form
000100 101001	I	..XX			vmsumshs	v2.03			344	Vector Multiply-Sum Signed Halfword Saturate VA-form
000100 101010	I	..XX			vsel	v2.03			287	Vector Select VA-form
000100 101011	I	..XX			vperm	v2.03			286	Vector Permute VA-form
000100 .... /.... 101100	I	..XX			vsldoi	v2.03			289	Vector Shift Left Double by Octet Immediate VA-form
000100 101101	I	..XX			vpermxor	v2.07			432	Vector Permute & Exclusive-OR VA-form
000100 101110	I	..XX			vmaddfp	v2.03			412	Vector Multiply-Add Floating-Point VA-form
000100 101111	I	..XX			vnmsubfp	v2.03			412	Vector Negative Multiply-Subtract Floating-Point VA-form
000100 110000	I	..XX			maddhd	v3.0			86	Multiply-Add High Doubleword VA-form
000100 110001	I	..XX			maddhdu	v3.0			86	Multiply-Add High Doubleword Unsigned VA-form
000100 110011	I	..XX			maddld	v3.0			86	Multiply-Add Low Doubleword VA-form
000100 111011	I	..XX			vpermr	v3.0			286	Vector Permute Right-indexed VA-form
000100 111100	I	..XX			vaddeuqm	v2.07			319	Vector Add Extended Unsigned Quadword Modulo VA-form
000100 111101	I	..XX			vaddecuq	v2.07			320	Vector Add Extended & write Carry-out Unsigned Quadword VA-form
000100 111110	I	..XX			vsubeuqm	v2.07			327	Vector Subtract Extended Unsigned Quadword Modulo VA-form
000100 111111	I	..XX			vsubecuq	v2.07			328	Vector Subtract Extended & write Carry-out Unsigned Quadword VA-form
000110 0000	I	..XX			lxvp	v3.1			603	Load VSX Vector Paired DQ-form
000110 0001	I	..XX			stxvp	v3.1			605	Store VSX Vector Paired DQ-form
000111	I	XXXX			mulli	P1			79	Multiply Low Immediate D-form
001000	I	XXXX			subfic	P1		SR	75	Subtract From Immediate Carrying D-form
001010 .../.	I	XXXX			cmpli	P1			91	Compare Logical Immediate D-form
001011 .../.	I	XXXX			cmpi	P1			90	Compare Immediate D-form
001100	I	XXXX			addic	P1		SR	75	Add Immediate Carrying D-form
001101	I	XXXX			addic.	P1		SR	75	Add Immediate Carrying and Record D-form
001110	I	XXXX			addi	P1			73	Add Immediate D-form
000001 100// .//.. .... 001110	I	..XX			paddi	v3.1			73	Prefixed Add Immediate MLS:D-form
001111	I	XXXX			addis	P1			74	Add Immediate Shifted D-form
010000	I	XXXX			bc[l][a]	P1		CT	40	Branch Conditional B-form
010001 ///// ///// ////. ..... .///01	I	XXXX			scv	v3.0			45	System Call Vectored SC-form
010001 ///// ///// ////. ..... .///1/	I	XXXX			sc	PPC			45	System Call SC-form
010010	I	XXXX			b[l][a]	P1			40	Branch I-form
010011 ...// ...// ///// 00000 00000/	I	XXXX			mcrf	P1			45	Move Condition Register Field XL-form
010011 .... 00001 00001/	I	XXXX			crnor	P1			44	Condition Register NOR XL-form
010011 .... 00100 00001/	I	XXXX			crandc	P1			44	Condition Register AND with Complement XL-form
010011 .... 00110 00001/	I	XXXX			crxor	P1			43	Condition Register XOR XL-form
010011 .... 00111 00001/	I	XXXX			crnand	P1			43	Condition Register NAND XL-form
010011 .... 01000 00001/	I	XXXX			crand	P1			43	Condition Register AND XL-form

Table E.1: Power ISA Instruction Set Sorted by Opcode (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
010011 .... 01001 00001/	I	XXXX			creqv	P1			44	Condition Register Equivalent XL-form
010011 .... 01101 00001/	I	XXXX			crorc	P1			44	Condition Register OR with Complement XL-form
010011 .... 01110 00001/	I	XXXX			cror	P1			43	Condition Register OR XL-form
010011 .... 00010.	I	XXXX			addpcis	v3.0			74	Add PC Immediate Shifted DX-form
010011 .... ///.. 00000 10000.	I	XXXX			bclr[l]	P1		CT	41	Branch Conditional to Link Register XL-form
010011 .... ///.. 10000 10000.	I	XXXX			bcctr[l]	P1		CT	41	Branch Conditional to Count Register XL-form
010011 .... ///.. 10001 10000.	I	XXXX			bctar[l]	v2.07			42	Branch Conditional to Branch Target Address Register XL-form
010011 ///// ///// ///// 00000 10010/	III	XXXX			rfid	PPC	P		1086	Return From Interrupt Doubleword XL-form
010011 ///// ///// ///// 00010 10010/	III	XXXX			rfscv	v3.0	P		1085	Return From System Call Vectored XL-form
010011 ///// ///// ////. 00100 10010/	I	...X	EBB		rfebb	v2.07			1037	Return from Event-Based Branch XL-form
010011 ///// ///// ///// 01000 10010/	III	..XX			hrfid	v2.02	HV		1086	Hypervisor Return From Interrupt Doubleword XL-form
010011 ///// ///// ///// 01001 10010/	III	....			urfid	v3.0C	UV		1087	Ultravisor Return From Interrupt Doubleword XL-form
010011 ///// ///// ///// 01011 10010/	III	..XX			stop	v3.0	P		1089	Stop XL-form
010011 ///// ///// ///// 00100 10110/	II	XXXX			isync	P1			1014	Instruction Synchronize XL-form
010100	I	XXXX			rlwimi[.]	P1		SR	109	Rotate Left Word Immediate then Mask Insert M-form
010101	I	XXXX			rlwinm[.]	P1		SR	108	Rotate Left Word Immediate then AND with Mask M-form
010111	I	XXXX			rlwnm[.]	P1		SR	109	Rotate Left Word then AND with Mask M-form
011000	I	XXXX			ori	P1			98	OR Immediate D-form
011001	I	XXXX			oris	P1			98	OR Immediate Shifted D-form
011010	I	XXXX			xori	P1			98	XOR Immediate D-form
011011	I	XXXX			xoris	P1			98	XOR Immediate Shifted D-form
011100	I	XXXX			andi.	P1		SR	97	AND Immediate D-form
011101	I	XXXX			andis.	P1		SR	97	AND Immediate Shifted D-form
011110 .... .000..	I	..XX			rldicl[.]	PPC		SR	110	Rotate Left Doubleword Immediate then Clear Left MD-form
011110 .... .001..	I	..XX			rldicr[.]	PPC		SR	110	Rotate Left Doubleword Immediate then Clear Right MD-form
011110 .... .010..	I	..XX			rldic[.]	PPC		SR	111	Rotate Left Doubleword Immediate then Clear MD-form
011110 .... .011..	I	..XX			rldimi[.]	PPC		SR	112	Rotate Left Doubleword Immediate then Mask Insert MD-form
011110 .... .1000.	I	..XX			rldcl[.]	PPC		SR	111	Rotate Left Doubleword then Clear Left MDS-form
011110 .... .1001.	I	..XX			rldcr[.]	PPC		SR	112	Rotate Left Doubleword then Clear Right MDS-form
011111 .../. 00000 00000/	I	XXXX			cmp	P1			90	Compare X-form
011111 .../. 00001 00000/	I	XXXX			cmpl	P1			91	Compare Logical X-form
011111 ..../// ///// 00100 00000/	I	XXXX			setb	v3.0			131	Set Boolean X-form
011111 .../. 00110 00000/	I	XXXX			cmprb	v3.0			92	Compare Ranged Byte X-form
011111 .../// 00111 00000/	I	XXXX			cmpeqb	v3.0			93	Compare Equal Byte X-form
011111 .... ///// 01100 00000/	I	XXXX			setbc	v3.1			131	Set Boolean Condition X-form
011111 .... ///// 01101 00000/	I	XXXX			setbcr	v3.1			131	Set Boolean Condition Reverse X-form
011111 .... ///// 01110 00000/	I	XXXX			setnbc	v3.1			131	Set Negative Boolean Condition X-form
011111 .... ///// 01111 00000/	I	XXXX			setnbcr	v3.1			131	Set Negative Boolean Condition Reverse X-form
011111 .../// ///// ///// 10010 00000/	I	XXXX			mcrxrx	v3.0			129	Move to CR from XER Extended X-form
011111 .... 00000 00100/	I	XXXX			tw	P1			94	Trap Word X-form
011111 .... 00010 00100/	I	..XX			td	PPC			95	Trap Doubleword X-form
011111 .... 00000 00110/	I	..XX			lvsl	v2.03			267	Load Vector for Shift Left Indexed X-form
011111 .... 00001 00110/	I	..XX			lvsr	v2.03			267	Load Vector for Shift Right Indexed X-form
011111 .... 10010 00110/	II	...X	AMO		lwat	v3.0			1011	Load Word Atomic X-form

Table E.1: Power ISA Instruction Set Sorted by Opcode (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
011111 .... 10011 00110/	II	...X	AMO		ldat	v3.0			1011	Load Doubleword Atomic X-form
011111 .... 10110 00110/	II	...X	AMO		stwat	v3.0			1013	Store Word Atomic X-form
011111 .... 10111 00110/	II	...X	AMO		stdat	v3.0			1013	Store Doubleword Atomic X-form
011111 ////1 .... 11000 00110/	II	....			copy	v3.0			1007	Copy X-form
011111 ///// ///// ///// 11010 00110/	II	....			cpabort	v3.0			1008	Copy-Paste Abort X-form
011111 ////. .... 11100 00110.	II	....			paste.	v3.0			1008	Paste X-form
011111 .... 00000 00111/	I	..XX			lvebx	v2.03			260	Load Vector Element Byte Indexed X-form
011111 .... 00001 00111/	I	..XX			lvehx	v2.03			260	Load Vector Element Halfword Indexed X-form
011111 .... 00010 00111/	I	..XX			lvewx	v2.03			261	Load Vector Element Word Indexed X-form
011111 .... 00011 00111/	I	..XX			lvx	v2.03			262	Load Vector Indexed X-form
011111 .... 00100 00111/	I	..XX			stvebx	v2.03			263	Store Vector Element Byte Indexed X-form
011111 .... 00101 00111/	I	..XX			stvehx	v2.03			264	Store Vector Element Halfword Indexed X-form
011111 .... 00110 00111/	I	..XX			stvewx	v2.03			264	Store Vector Element Word Indexed X-form
011111 .... 00111 00111/	I	..XX			stvx	v2.03			265	Store Vector Indexed X-form
011111 .... 01011 00111/	I	..XX			lvxl	v2.03			262	Load Vector Indexed Last X-form
011111 .... 01111 00111/	I	..XX			stvxl	v2.03			265	Store Vector Indexed Last X-form
011111 .... 00000 01000.	I	XXXX			subfc[.]	P1		SR	76	Subtract From Carrying XO-form
011111 .... 00001 01000.	I	XXXX			subf[.]	PPC		SR	75	Subtract From XO-form
011111 .... ///// 00011 01000.	I	XXXX			neg[.]	P1		SR	78	Negate XO-form
011111 .... 00100 01000.	I	XXXX			subfe[.]	P1		SR	76	Subtract From Extended XO-form
011111 .... ///// 00110 01000.	I	XXXX			subfze[.]	P1		SR	77	Subtract From Zero Extended XO-form
011111 .... ///// 00111 01000.	I	XXXX			subfme[.]	P1		SR	77	Subtract From Minus One Extended XO-form
011111 .... 10000 01000.	I	..XX			subfco[.]	P1		SR	76	Subtract From Carrying & record OV XO-form
011111 .... 10001 01000.	I	..XX			subfo[.]	PPC		SR	75	Subtract From & record OV XO-form
011111 .... ///// 10011 01000.	I	..XX			nego[.]	P1		SR	78	Negate & record OV XO-form
011111 .... 10100 01000.	I	..XX			subfeo[.]	P1		SR	76	Subtract From Extended & record OV XO-form
011111 .... ///// 10110 01000.	I	..XX			subfzeo[.]	P1		SR	77	Subtract From Zero Extended & record OV XO-form
011111 .... ///// 10111 01000.	I	..XX			subfmeo[.]	P1		SR	77	Subtract From Minus One Extended & record OV XO-form
011111 .... /0000 01001.	I	..XX			mulhdu[.]	PPC		SR	85	Multiply High Doubleword Unsigned XO-form
011111 .... /0010 01001.	I	..XX			mulhd[.]	PPC		SR	85	Multiply High Doubleword XO-form
011111 .... 00111 01001.	I	..XX			mulld[.]	PPC		SR	85	Multiply Low Doubleword XO-form
011111 .... 01000 01001/	I	..XX			modud	v3.0			89	Modulo Unsigned Doubleword X-form
011111 .... 01100 01001.	I	..XX			divdeu[.]	v2.06		SR	88	Divide Doubleword Extended Unsigned XO-form
011111 .... 01101 01001.	I	..XX			divde[.]	v2.06		SR	88	Divide Doubleword Extended XO-form
011111 .... 01110 01001.	I	..XX			divdu[.]	PPC		SR	87	Divide Doubleword Unsigned XO-form
011111 .... 01111 01001.	I	..XX			divd[.]	PPC		SR	87	Divide Doubleword XO-form
011111 .... 10111 01001.	I	..XX			mulldo[.]	PPC		SR	85	Multiply Low Doubleword & record OV XO-form
011111 .... 11000 01001/	I	..XX			modsd	v3.0			89	Modulo Signed Doubleword X-form
011111 .... 11100 01001.	I	..XX			divdeuo[.]	v2.06		SR	88	Divide Doubleword Extended Unsigned & record OV XO-form
011111 .... 11101 01001.	I	..XX			divdeo[.]	v2.06		SR	88	Divide Doubleword Extended & record OV XO-form
011111 .... 11110 01001.	I	..XX			divduo[.]	PPC		SR	87	Divide Doubleword Unsigned & record OV XO-form
011111 .... 11111 01001.	I	..XX			divdo[.]	PPC		SR	87	Divide Doubleword & record OV XO-form
011111 ....101 01010/	I	XXXX			addex	v3.0B			78	Add Extended using alternate carry bit Z23-form
011111 .... /0010 01010/	I	XXXX			addg6s	v2.06			117	Add and Generate Sixes XO-form
011111 .... 00000 01010.	I	XXXX			addc[.]	P1		SR	76	Add Carrying XO-form
011111 .... 00100 01010.	I	XXXX			adde[.]	P1		SR	76	Add Extended XO-form
011111 .... ///// 00110 01010.	I	XXXX			addze[.]	P1		SR	77	Add to Zero Extended XO-form

Table E.1: Power ISA Instruction Set Sorted by Opcode (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
011111 .... ///// 00111 01010.	I	XXXX			addme[.]	P1		SR	77	Add to Minus One Extended XO-form
011111 .... 01000 01010.	I	XXXX			add[.]	P1		SR	74	Add XO-form
011111 .... 10000 01010.	I	..XX			addco[.]	P1		SR	76	Add Carrying & record OV XO-form
011111 .... 10100 01010.	I	..XX			addeo[.]	P1		SR	76	Add Extended & record OV XO-form
011111 .... ///// 10110 01010.	I	..XX			addzeo[.]	P1		SR	77	Add to Zero Extended & record OV XO-form
011111 .... ///// 10111 01010.	I	..XX			addmeo[.]	P1		SR	77	Add to Minus One Extended & record OV XO-form
011111 .... 11000 01010.	I	..XX			addo[.]	P1		SR	74	Add & record OV XO-form
011111 .... /0000 01011.	I	XXXX			mulhwu[.]	PPC		SR	79	Multiply High Word Unsigned XO-form
011111 .... /0010 01011.	I	XXXX			mulhw[.]	PPC		SR	79	Multiply High Word XO-form
011111 .... 00111 01011.	I	XXXX			mullw[.]	P1		SR	79	Multiply Low Word XO-form
011111 .... 01000 01011/	I	XXXX			moduw	v3.0			83	Modulo Unsigned Word X-form
011111 .... 01100 01011.	I	XXXX			divweu[.]	v2.06		SR	81	Divide Word Extended Unsigned XO-form
011111 .... 01101 01011.	I	XXXX			divwe[.]	v2.06		SR	81	Divide Word Extended XO-form
011111 .... 01110 01011.	I	XXXX			divwu[.]	PPC		SR	80	Divide Word Unsigned XO-form
011111 .... 01111 01011.	I	XXXX			divw[.]	PPC		SR	80	Divide Word XO-form
011111 .... 10111 01011.	I	..XX			mullwo[.]	P1		SR	79	Multiply Low Word & record OV XO-form
011111 .... 11000 01011/	I	XXXX			modsw	v3.0			83	Modulo Signed Word X-form
011111 .... 11100 01011.	I	..XX			divweuo[.]	v2.06		SR	81	Divide Word Extended Unsigned & record OV XO-form
011111 .... 11101 01011.	I	..XX			divweo[.]	v2.06		SR	81	Divide Word Extended & record OV XO-form
011111 .... 11110 01011.	I	..XX			divwuo[.]	PPC		SR	80	Divide Word Unsigned & record OV XO-form
011111 .... 11111 01011.	I	..XX			divwo[.]	PPC		SR	80	Divide Word & record OV XO-form
011111 .... 00000 01100.	I	..XX			lxsiwzx	v2.07			566	Load VSX Scalar as Integer Word & Zero Indexed X-form
011111 .... 00010 01100.	I	..XX			lxsiwax	v2.07			565	Load VSX Scalar as Integer Word Algebraic Indexed X-form
011111 .... 00100 01100.	I	..XX			stxsiwx	v2.07			572	Store VSX Scalar as Integer Word Indexed X-form
011111 .... 0100/ 01100.	I	..XX			lxvx	v3.0			580	Load VSX Vector Indexed X-form
011111 .... 01010 01100.	I	..XX			lxvdsx	v2.06			582	Load VSX Vector Doubleword & Splat Indexed X-form
011111 .... 01011 01100.	I	..XX			lxvwsx	v3.0			583	Load VSX Vector Word & Splat Indexed X-form
011111 .... 01100 01100.	I	..XX			stxvx	v3.0			596	Store VSX Vector Indexed X-form
011111 .... 10000 01100.	I	..XX			lxsspx	v2.07			568	Load VSX Scalar Single-Precision Indexed X-form
011111 .... 10010 01100.	I	..XX			lxsdx	v2.06			563	Load VSX Scalar Doubleword Indexed X-form
011111 .... 10100 01100.	I	..XX			stxsspx	v2.07			574	Store VSX Scalar Single-Precision Indexed X-form
011111 .... 10110 01100.	I	..XX			stxsdx	v2.06			570	Store VSX Scalar Doubleword Indexed X-form
011111 .... 11000 01100.	I	..XX			lxvw4x	v2.06			579	Load VSX Vector Word*4 Indexed X-form
011111 .... 11001 01100.	I	..XX			lxvh8x	v3.0			578	Load VSX Vector Halfword*8 Indexed X-form
011111 .... 11010 01100.	I	..XX			lxvd2x	v2.06			577	Load VSX Vector Doubleword*2 Indexed X-form
011111 .... 11011 01100.	I	..XX			lxvb16x	v3.0			576	Load VSX Vector Byte*16 Indexed X-form
011111 .... 11100 01100.	I	..XX			stxvw4x	v2.06			595	Store VSX Vector Word*4 Indexed X-form
011111 .... 11101 01100.	I	..XX			stxvh8x	v3.0			594	Store VSX Vector Halfword*8 Indexed X-form
011111 .... 11110 01100.	I	..XX			stxvd2x	v2.06			593	Store VSX Vector Doubleword*2 Indexed X-form
011111 .... 11111 01100.	I	..XX			stxvb16x	v3.0			592	Store VSX Vector Byte*16 Indexed X-form
011111 .... 00000 01101.	I	..XX			lxvrbx	v3.1			584	Load VSX Vector Rightmost Byte Indexed X-form
011111 .... 00001 01101.	I	..XX			lxvrhx	v3.1			586	Load VSX Vector Rightmost Halfword Indexed X-form
011111 .... 00010 01101.	I	..XX			lxvrwx	v3.1			587	Load VSX Vector Rightmost Word Indexed X-form
011111 .... 00011 01101.	I	..XX			lxvrdx	v3.1			585	Load VSX Vector Rightmost Doubleword Indexed X-form
011111 .... 00100 01101.	I	..XX			stxvrbx	v3.1			598	Store VSX Vector Rightmost Byte Indexed X-form
011111 .... 00101 01101.	I	..XX			stxvrhx	v3.1			599	Store VSX Vector Rightmost Halfword Indexed X-form
011111 .... 00110 01101.	I	..XX			stxvrwx	v3.1			599	Store VSX Vector Rightmost Word Indexed X-form

Table E.1: Power ISA Instruction Set Sorted by Opcode (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
011111 .... 00111 01101.	I	..XX			stxvrdx	v3.1			598	Store VSX Vector Rightmost Doubleword Indexed X-form
011111 .... 01000 01101.	I	..XX			lxvl	v3.0			588	Load VSX Vector with Length X-form
011111 .... 01001 01101.	I	..XX			lxvll	v3.0			590	Load VSX Vector with Length Left-justified X-form
011111 .... 01010 01101/	I	..XX			lxvpx	v3.1			604	Load VSX Vector Paired Indexed X-form
011111 .... 01100 01101.	I	..XX			stxvl	v3.0			600	Store VSX Vector with Length X-form
011111 .... 01101 01101.	I	..XX			stxvll	v3.0			602	Store VSX Vector with Length Left-justified X-form
011111 .... 01110 01101/	I	..XX			stxvpx	v3.1			606	Store VSX Vector Paired Indexed X-form
011111 .... 11000 01101.	I	..XX			lxsibzx	v3.0			564	Load VSX Scalar as Integer Byte & Zero Indexed X-form
011111 .... 11001 01101.	I	..XX			lxsihzx	v3.0			564	Load VSX Scalar as Integer Halfword & Zero Indexed X-form
011111 .... 11100 01101.	I	..XX			stxsibx	v3.0			571	Store VSX Scalar as Integer Byte Indexed X-form
011111 .... 11101 01101.	I	..XX			stxsihx	v3.0			571	Store VSX Scalar as Integer Halfword Indexed X-form
011111 ///// ///// 00010 01110/	III	....			msgsndu	v3.0C	UV		1274	Message Send Ultravisor X-form
011111 ///// ///// 00011 01110/	III	....			msgclru	v3.0C	UV		1274	Message Clear Ultravisor X-form
011111 ///// ///// 00100 01110/	III	..XX			msgsndp	v2.07	P		1276	Message Send Privileged X-form
011111 ///// ///// 00101 01110/	III	..XX			msgclrp	v2.07	P		1276	Message Clear Privileged X-form
011111 ///// ///// 00110 01110/	III	..XX			msgsnd	v2.07	HV		1275	Message Send X-form
011111 ///// ///// 00111 01110/	III	..XX			msgclr	v2.07	HV		1275	Message Clear X-form
011111 .... 01001 01110/	I	...X	BHRB		mfbhrbe	v2.07			1041	Move From Branch History Rolling Buffer Entry XFX-form
011111 ///// ///// ///// 01101 01110/	I	...X	BHRB		clrbhrb	v2.07			1041	Clear BHRB X-form
011111 .... 01111/	I	XXXX			isel	v2.03			96	Integer Select A-form
011111 ..... 0.... ..../ 00100 10000/	I	XXXX			mtcrf	P1			129	Move To Condition Register Fields XFX-form
011111 ..... 1.... ..../ 00100 10000/	I	XXXX			mtocrf	v2.01			129	Move To One Condition Register Field XFX-form
011111 .../// 00000 ///// 00101 10001/	I	....	MMA	MMA	xxmfacc	v3.1			876	VSX Move From Accumulator X-form
011111 .../// 00001 ///// 00101 10001/	I	....	MMA	MMA	xxmtacc	v3.1			877	VSX Move To Accumulator X-form
011111 .../// 00011 ///// 00101 10001/	I	....	MMA	MMA	xxsetaccz	v3.1			877	VSX Set Accumulator to Zero X-form
011111 ..... ////. ///// 00100 10010/	III	XXXX			mtmsr	P1	P		1108	Move To Machine State Register X-form
011111 ..... ////. ///// 00101 10010/	III	..XX			mtmsrd	PPC	P		1109	Move To Machine State Register Doubleword X-form
011111 ..... /.... ..... 01000 10010/	III	..XX			tlbiel	v2.03	P	64	1181	TLB Invalidate Entry Local X-form
011111 ..... /.... ..... 01001 10010/	III	...X			tlbie	P1	HV	64	1174	TLB Invalidate Entry X-form
011111 ///// ///// ///// 01010 10010/	III	...X			slbsync	v3.0	P		1171	SLB Synchronize X-form
011111 ..... ///// 01100 10010/	III	...X			slbmte	v2.00	P		1168	SLB Move To Entry X-form
011111 ///// ///// 01101 10010/	III	...X			slbie	PPC	P		1160	SLB Invalidate Entry X-form
011111 ..... ///// 01110 10010/	III	...X			slbieg	v3.0	P		1162	SLB Invalidate Entry Global X-form
011111 //... ///// ///// 01111 10010/	III	..XX			slbia	PPC	P		1164	SLB Invalidate All X-form
011111 .... 10100 10010.	III	....			hashstp	v3.1B	P		1099	Hash Store Privileged X-form
011111 .... 10101 10010.	III	....			hashchkp	v3.1B	P		1099	Hash Check Privileged X-form
011111 .... 10110 10010.	I	....			hashst	v3.1B			120	Hash Store X-form
011111 .... 10111 10010.	I	....			hashchk	v3.1B			120	Hash Check X-form
011111 ..... ////. ///// 11010 10010/	III	...X			slbiag	v3.0B	P		1167	SLB Invalidate All Global X-form
011111 ..... 0//// ///// 00000 10011/	I	XXXX			mfcr	P1			130	Move From Condition Register XFX-form
011111 ..... 1.... ..../ 00000 10011/	I	XXXX			mfocrf	v2.01			130	Move From One Condition Register Field XFX-form
011111 .... ///// 00001 10011.	I	..XX			mfvsrd	v2.07			122	Move From VSR Doubleword X-form
011111 ..... ///// ///// 00010 10011/	III	XXXX			mfmsr	P1	P		1110	Move From Machine State Register X-form
011111 .... ///// 00011 10011.	I	..XX			mfvsrwz	v2.07			123	Move From VSR Word and Zero X-form
011111 .... ///// 00101 10011.	I	..XX			mtvsrd	v2.07			123	Move To VSR Doubleword X-form
011111 .... ///// 00110 10011.	I	..XX			mtvsrwa	v2.07			124	Move To VSR Word Algebraic X-form
011111 .... ///// 00111 10011.	I	..XX			mtvsrwz	v2.07			124	Move To VSR Word and Zero X-form

Table E.1: Power ISA Instruction Set Sorted by Opcode (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
011111 .... ///// 01001 10011.	I	..XX			mfvsrld	v3.0			122	Move From VSR Lower Doubleword X-form
011111 .... 01010 10011/	I III	XXXX			mfspr	P1	O		128	Move From Special Purpose Register XFX-form
011111 .... 01011 10011/	II	XXXX			mftb	PPC			1032	Move From Time Base XFX-form
011111 .... ///// 01100 10011.	I	..XX			mtvsrws	v3.0			125	Move To VSR Word & Splat X-form
011111 .... 01101 10011.	I	..XX			mtvsrdd	v3.0			125	Move To VSR Double Doubleword X-form
011111 .... 01110 10011/	I III	XXXX			mtspr	P1	O		126	Move To Special Purpose Register XFX-form
011111 ..... ///.. ///// 10111 10011/	I	XXXX			darn	v3.0			84	Deliver A Random Number X-form
011111 ..... ///// 11010 10011/	III	...X			slbmfev	v2.00	P		1169	SLB Move From Entry VSID X-form
011111 ..... ///// 11100 10011/	III	...X			slbmfee	v2.00	P		1169	SLB Move From Entry ESID X-form
011111 ..... ///// 11110 100111	III	...X			slbfee.	v2.05	P	SR	1170	SLB Find Entry ESID X-form
011111 .... 00000 10100.	II	XXXX			lwarx	PPC			1017	Load Word And Reserve Indexed X-form
011111 .... 00001 10100.	II	XXXX			lbarx	v2.06			1016	Load Byte And Reserve Indexed X-form
011111 .... 00010 10100.	II	..XX			ldarx	PPC			1021	Load Doubleword And Reserve Indexed X-form
011111 .... 00011 10100.	II	XXXX			lharx	v2.06			1016	Load Halfword And Reserve Indexed X-form
011111 .... 01000 10100.	I	..XX			lqarx	v2.07			1023	Load Quadword And Reserve Indexed X-form
011111 .... 10000 10100/	I	..XX			ldbrx	v2.06			67	Load Doubleword Byte-Reverse Indexed X-form
011111 .... 10100 10100/	I	..XX			stdbrx	v2.06			67	Store Doubleword Byte-Reverse Indexed X-form
011111 .... 00000 10101/	I	..XX			ldx	PPC			55	Load Doubleword Indexed X-form
011111 .... 00001 10101/	I	..XX			ldux	PPC			55	Load Doubleword with Update Indexed X-form
011111 .... 00100 10101/	I	..XX			stdx	PPC			60	Store Doubleword Indexed X-form
011111 .... 00101 10101/	I	..XX			stdux	PPC			61	Store Doubleword with Update Indexed X-form
011111 .... 01010 10101/	I	..XX			lwax	PPC			54	Load Word Algebraic Indexed X-form
011111 .... 01011 10101/	I	..XX			lwaux	PPC			54	Load Word Algebraic with Update Indexed X-form
011111 .... 10000 10101/	I	...X			lswx	P1			70	Load String Word Indexed X-form
011111 .... 10010 10101/	I	...X			lswi	P1			70	Load String Word Immediate X-form
011111 .... 10100 10101/	I	...X			stswx	P1			71	Store String Word Indexed X-form
011111 .... 10110 10101/	I	...X			stswi	P1			71	Store String Word Immediate X-form
011111 .... 11000 10101/	III	..XX			lwzcix	v2.05	HV		1097	Load Word and Zero Caching Inhibited Indexed X-form
011111 .... 11001 10101/	III	..XX			lhzcix	v2.05	HV		1097	Load Halfword and Zero Caching Inhibited Indexed X-form
011111 .... 11010 10101/	III	..XX			lbzcix	v2.05	HV		1097	Load Byte and Zero Caching Inhibited Indexed X-form
011111 .... 11011 10101/	III	..XX			ldcix	v2.05	HV		1097	Load Doubleword Caching Inhibited Indexed X-form
011111 .... 11100 10101/	III	..XX			stwcix	v2.05	HV		1098	Store Word Caching Inhibited Indexed X-form
011111 .... 11101 10101/	III	..XX			sthcix	v2.05	HV		1098	Store Halfword Caching Inhibited Indexed X-form
011111 .... 11110 10101/	III	..XX			stbcix	v2.05	HV		1098	Store Byte Caching Inhibited Indexed X-form
011111 .... 11111 10101/	III	..XX			stdcix	v2.05	HV		1098	Store Doubleword Caching Inhibited Indexed X-form
011111 /.... .... 00000 10110/	II	..XX			icbt	v2.07			991	Instruction Cache Block Touch X-form
011111 ///// 00001 10110/	II	..XX			dcbst	PPC			1002	Data Cache Block Store X-form
011111 //... .... 00010 10110/	II	..XX			dcbf	PPC			1003	Data Cache Block Flush X-form
011111 .... 00111 10110/	II	..XX			dcbtst	PPC			1000	Data Cache Block Touch for Store X-form
011111 .... 01000 10110/	II	..XX			dcbt	PPC			999	Data Cache Block Touch X-form
011111 .... 10000 10110/	I	XXXX			lwbrx	P1			66	Load Word Byte-Reverse Indexed X-form
011111 ///// ///// ///// 10001 10110/	III	...X			tlbsync	PPC	HV/P		1186	TLB Synchronize X-form
011111 //... ///.. ///// 10010 10110/	II	XXXX			sync	P1			1025	Synchronize X-form
011111 .... 10100 10110/	I	XXXX			stwbrx	P1			66	Store Word Byte-Reverse Indexed X-form
011111 .... 11000 10110/	I	XXXX			lhbrx	P1			66	Load Halfword Byte-Reverse Indexed X-form
011111 ///// ///// ///// 11010 10110/	II	XXXX			eieio	PPC			1028	Enforce In-order Execution of I/O X-form

Table E.1: Power ISA Instruction Set Sorted by Opcode (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
011111 ///// ///// ///// 11011 10110/	III	..XX			msgsync	v3.0	HV		1277	Message Synchronize X-form
011111 .... 11100 10110/	I	XXXX			sthbrx	P1			66	Store Halfword Byte-Reverse Indexed X-form
011111 ///// 11110 10110/	II	..XX			icbi	PPC			991	Instruction Cache Block Invalidate X-form
011111 ///// 11111 10110/	II	XXXX			dcbz	P1			1001	Data Cache Block set to Zero X-form
011111 .... 00100 101101	II	XXXX			stwcx.	PPC			1020	Store Word Conditional Indexed X-form
011111 .... 00101 101101	I	..XX			stqcx.	v2.07			1024	Store Quadword Conditional Indexed X-form
011111 .... 00110 101101	II	..XX			stdcx.	PPC			1022	Store Doubleword Conditional Indexed X-form
011111 .... 10101 101101	II	XXXX			stbcx.	v2.06			1018	Store Byte Conditional Indexed X-form
011111 .... 10110 101101	II	XXXX			sthcx.	v2.06			1019	Store Halfword Conditional Indexed X-form
011111 .... 00000 10111/	I	XXXX			lwzx	P1			53	Load Word and Zero Indexed X-form
011111 .... 00001 10111/	I	XXXX			lwzux	P1			53	Load Word and Zero with Update Indexed X-form
011111 .... 00010 10111/	I	XXXX			lbzx	P1			48	Load Byte and Zero Indexed X-form
011111 .... 00011 10111/	I	XXXX			lbzux	P1			49	Load Byte and Zero with Update Indexed X-form
011111 .... 00100 10111/	I	XXXX			stwx	P1			59	Store Word Indexed X-form
011111 .... 00101 10111/	I	XXXX			stwux	P1			59	Store Word with Update Indexed X-form
011111 .... 00110 10111/	I	XXXX			stbx	P1			57	Store Byte Indexed X-form
011111 .... 00111 10111/	I	XXXX			stbux	P1			57	Store Byte with Update Indexed X-form
011111 .... 01000 10111/	I	XXXX			lhzx	P1			50	Load Halfword and Zero Indexed X-form
011111 .... 01001 10111/	I	XXXX			lhzux	P1			51	Load Halfword and Zero with Update Indexed X-form
011111 .... 01010 10111/	I	XXXX			lhax	P1			52	Load Halfword Algebraic Indexed X-form
011111 .... 01011 10111/	I	XXXX			lhaux	P1			52	Load Halfword Algebraic with Update Indexed X-form
011111 .... 01100 10111/	I	XXXX			sthx	P1			58	Store Halfword Indexed X-form
011111 .... 01101 10111/	I	XXXX			sthux	P1			58	Store Halfword with Update Indexed X-form
011111 .... 10000 10111/	I	.XXX			lfsx	P1			150	Load Floating-Point Single Indexed X-form
011111 .... 10001 10111/	I	.XXX			lfsux	P1			151	Load Floating-Point Single with Update Indexed X-form
011111 .... 10010 10111/	I	.XXX			lfdx	P1			152	Load Floating-Point Double Indexed X-form
011111 .... 10011 10111/	I	.XXX			lfdux	P1			153	Load Floating-Point Double with Update Indexed X-form
011111 .... 10100 10111/	I	.XXX			stfsx	P1			155	Store Floating-Point Single Indexed X-form
011111 .... 10101 10111/	I	.XXX			stfsux	P1			155	Store Floating-Point Single with Update Indexed X-form
011111 .... 10110 10111/	I	.XXX			stfdx	P1			156	Store Floating-Point Double Indexed X-form
011111 .... 10111 10111/	I	.XXX			stfdux	P1			157	Store Floating-Point Double with Update Indexed X-form
011111 .... 11000 10111/	I	...X			lfdpx	v2.05			158	Load Floating-Point Double Pair Indexed X-form
011111 .... 11010 10111/	I	.XXX			lfiwax	v2.05			153	Load Floating-Point as Integer Word Algebraic Indexed X-form
011111 .... 11011 10111/	I	.XXX			lfiwzx	v2.06			153	Load Floating-Point as Integer Word & Zero Indexed X-form
011111 .... 11100 10111/	I	...X			stfdpx	v2.05			159	Store Floating-Point Double Pair Indexed X-form
011111 .... 11110 10111/	I	.XXX			stfiwx	PPC			157	Store Floating-Point as Integer Word Indexed X-form
011111 .... 00000 11000.	I	XXXX			slw[.]	P1		SR	113	Shift Left Word X-form
011111 .... 10000 11000.	I	XXXX			srw[.]	P1		SR	113	Shift Right Word X-form
011111 .... 11000 11000.	I	XXXX			sraw[.]	P1		SR	114	Shift Right Algebraic Word X-form
011111 .... 11001 11000.	I	XXXX			srawi[.]	P1		SR	114	Shift Right Algebraic Word Immediate X-form
011111 .... 11001 1101..	I	..XX			sradi[.]	PPC		SR	115	Shift Right Algebraic Doubleword Immediate XS-form
011111 .... 11011 1101..	I	..XX			extswsli[.]	v3.0			116	Extend Sign Word and Shift Left Immediate XS-form
011111 .... ///// 00000 11010.	I	XXXX			cntlzw[.]	P1		SR	101	Count Leading Zeros Word X-form
011111 .... ///// 00001 11010.	I	..XX			cntlzd[.]	PPC		SR	104	Count Leading Zeros Doubleword X-form
011111 .... ///// 00011 11010/	I	XXXX			popcntb	v2.02			102	Population Count Bytes X-form
011111 .... ///// 00100 11010/	I	XXXX			prtyw	v2.05			102	Parity Word X-form

Table E.1: Power ISA Instruction Set Sorted by Opcode (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
011111 .... ///// 00101 11010/	I	..XX			prtyd	v2.05			103	Parity Doubleword X-form
011111 .... ///// 01000 11010/	I	XXXX			cdtbcd	v2.06			117	Convert Declets To Binary Coded Decimal X-form
011111 .... ///// 01001 11010/	I	XXXX			cbcdtd	v2.06			117	Convert Binary Coded Decimal To Declets X-form
011111 .... ///// 01011 11010/	I	XXXX			popcntw	v2.06			102	Population Count Words X-form
011111 .... ///// 01111 11010/	I	..XX			popcntd	v2.06			103	Population Count Doubleword X-form
011111 .... ///// 10000 11010.	I	XXXX			cnttzw[.]	v3.0			101	Count Trailing Zeros Word X-form
011111 .... ///// 10001 11010.	I	..XX			cnttzd[.]	v3.0			104	Count Trailing Zeros Doubleword X-form
011111 .... 11000 11010.	I	..XX			srad[.]	PPC		SR	115	Shift Right Algebraic Doubleword X-form
011111 .... ///// 11100 11010.	I	XXXX			extsh[.]	P1		SR	101	Extend Sign Halfword X-form
011111 .... ///// 11101 11010.	I	XXXX			extsb[.]	PPC		SR	101	Extend Sign Byte X-form
011111 .... ///// 11110 11010.	I	..XX			extsw[.]	PPC		SR	103	Extend Sign Word X-form
011111 .... 00000 11011.	I	..XX			sld[.]	PPC		SR	115	Shift Left Doubleword X-form
011111 .... 00001 11011/	I	..XX			cntlzdm	v3.1			104	Count Leading Zeros Doubleword under bit Mask X-form
011111 .... ///// 00100 11011/	I	XXXX			brw	v3.1			118	Byte-Reverse Word X-form
011111 .... ///// 00101 11011/	I	..XX			brd	v3.1			118	Byte-Reverse Doubleword X-form
011111 .... ///// 00110 11011/	I	XXXX			brh	v3.1			118	Byte-Reverse Halfword X-form
011111 .... 10000 11011.	I	..XX			srd[.]	PPC		SR	115	Shift Right Doubleword X-form
011111 .... 10001 11011/	I	..XX			cnttzdm	v3.1			104	Count Trailing Zeros Doubleword under bit Mask X-form
011111 .... 00000 11100.	I	XXXX			and[.]	P1		SR	99	AND X-form
011111 .... 00001 11100.	I	XXXX			andc[.]	P1		SR	100	AND with Complement X-form
011111 .... 00011 11100.	I	XXXX			nor[.]	P1		SR	100	NOR X-form
011111 .... 00100 11100/	I	..XX			pdepd	v3.1			106	Parallel Bits Deposit Doubleword X-form
011111 .... 00101 11100/	I	..XX			pextd	v3.1			106	Parallel Bits Extract Doubleword X-form
011111 .... 00110 11100/	I	..XX			cfuged	v3.1			105	Centrifuge Doubleword X-form
011111 .... 00111 11100/	I	..XX			bpermd	v2.06			105	Bit Permute Doubleword X-form
011111 .... 01000 11100.	I	XXXX			eqv[.]	P1		SR	100	Equivalent X-form
011111 .... 01001 11100.	I	XXXX			xor[.]	P1		SR	99	XOR X-form
011111 .... 01100 11100.	I	XXXX			orc[.]	P1		SR	100	OR with Complement X-form
011111 .... 01101 11100.	I	XXXX			or[.]	P1		SR	99	OR X-form
011111 .... 01110 11100.	I	XXXX			nand[.]	P1		SR	99	NAND X-form
011111 .... 01111 11100/	I	XXXX			cmpb	v2.05			101	Compare Bytes X-form
011111 ///.. ///.. ///// 00000 11110/	II	..XX			wait	v2.03			1029	Wait X-form
100000	I	XXXX			lwz	P1			53	Load Word and Zero D-form
000001 100// .//.. .... 100000	I	..XX			plwz	v3.1			53	Prefixed Load Word and Zero MLS:D-form
000001 01000 0//// .... 100000 ..... 000..	I	..XX			xxspltiw32dx	v3.1			919	VSX Vector Splat Immediate32 Doubleword Indexed 8RR:D-form
000001 01000 0//// .... 100000 ..... 0010.	I	..XX			xxspltidp	v3.1			920	VSX Vector Splat Immediate Double-Precision 8RR:D-form
000001 01000 0//// .... 100000 ..... 0011.	I	..XX			xxspltiw	v3.1			920	VSX Vector Splat Immediate Word 8RR:D-form
100001	I	XXXX			lwzu	P1			53	Load Word and Zero with Update D-form
000001 01000 0//// ///// ///// ////// 100001 .... 00....	I	..XX			xxblendvb	v3.1			912	VSX Vector Blend Variable Byte 8RR:XX4-form
000001 01000 0//// ///// ///// ////// 100001 .... 01....	I	..XX			xxblendvh	v3.1			913	VSX Vector Blend Variable Halfword 8RR:XX4-form
000001 01000 0//// ///// ///// ////// 100001 .... 10....	I	..XX			xxblendvw	v3.1			913	VSX Vector Blend Variable Word 8RR:XX4-form

Table E.1: Power ISA Instruction Set Sorted by Opcode (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
000001 01000 0//// ///// ///// ////// 100001 .... 11....	I	..XX			xxblendvd	v3.1			912	VSX Vector Blend Variable Doubleword 8RR:XX4-form
100010	I	XXXX			lbz	P1			48	Load Byte and Zero D-form
000001 100// .//.. .... 100010	I	..XX			plbz	v3.1			48	Prefixed Load Byte and Zero MLS:D-form
000001 01000 0//// ///// ///// ///... 100010 .... 00....	I	..XX			xxpermx	v3.1			924	VSX Vector Permute Extended 8RR:XX4-form
000001 01000 0//// ///// ///.. ..... 100010 .... 01....	I	..XX			xxeval	v3.1			910	VSX Vector Evaluate 8RR:XX4-form
100011	I	XXXX			lbzu	P1			49	Load Byte and Zero with Update D-form
100100	I	XXXX			stw	P1			59	Store Word D-form
000001 100// .//.. .... 100100	I	..XX			pstw	v3.1			59	Prefixed Store Word MLS:D-form
100101	I	XXXX			stwu	P1			59	Store Word with Update D-form
100110	I	XXXX			stb	P1			57	Store Byte D-form
000001 100// .//.. .... 100110	I	..XX			pstb	v3.1			57	Prefixed Store Byte MLS:D-form
100111	I	XXXX			stbu	P1			57	Store Byte with Update D-form
101000	I	XXXX			lhz	P1			50	Load Halfword and Zero D-form
000001 100// .//.. .... 101000	I	..XX			plhz	v3.1			50	Prefixed Load Halfword and Zero MLS:D-form
101001	I	XXXX			lhzu	P1			51	Load Halfword and Zero with Update D-form
000001 000// .//.. .... 101001	I	..XX			plwa	v3.1			54	Prefixed Load Word Algebraic 8LS:D-form
101010	I	XXXX			lha	P1			51	Load Halfword Algebraic D-form
000001 000// .//.. .... 101010	I	..XX			plxsd	v3.1			562	Prefixed Load VSX Scalar Doubleword 8LS:D-form
000001 100// .//.. .... 101010	I	..XX			plha	v3.1			51	Prefixed Load Halfword Algebraic MLS:D-form
101011	I	XXXX			lhau	P1			52	Load Halfword Algebraic with Update D-form
000001 000// .//.. .... 101011	I	..XX			plxssp	v3.1			567	Prefixed Load VSX Scalar Single-Precision 8LS:D-form
101100	I	XXXX			sth	P1			58	Store Halfword D-form
000001 100// .//.. .... 101100	I	..XX			psth	v3.1			58	Prefixed Store Halfword MLS:D-form
101101	I	XXXX			sthu	P1			58	Store Halfword with Update D-form
101110	I	...X			lmw	P1			68	Load Multiple Word D-form
000001 000// .//.. .... 101110	I	..XX			pstxsd	v3.1			569	Prefixed Store VSX Scalar Doubleword 8LS:D-form
101111	I	...X			stmw	P1			68	Store Multiple Word D-form
000001 000// .//.. .... 101111	I	..XX			pstxssp	v3.1			573	Prefixed Store VSX Scalar Single-Precision 8LS:D-form
110000	I	.XXX			lfs	P1			150	Load Floating-Point Single D-form
000001 100// .//.. .... 110000	I	..XX			plfs	v3.1			150	Prefixed Load Floating-Point Single MLS:D-form
110001	I	.XXX			lfsu	P1			151	Load Floating-Point Single with Update D-form
000001 000// .//.. .... 11001.	I	..XX			plxv	v3.1			575	Prefixed Load VSX Vector 8LS:D-form
110010	I	.XXX			lfd	P1			152	Load Floating-Point Double D-form

Table E.1: Power ISA Instruction Set Sorted by Opcode (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
000001 100// .//.. .... 110010	I	..XX			plfd	v3.1			152	Prefixed Load Floating-Point Double MLS:D-form
110011	I	.XXX			lfdu	P1			153	Load Floating-Point Double with Update D-form
110100	I	.XXX			stfs	P1			154	Store Floating-Point Single D-form
000001 100// .//.. .... 110100	I	..XX			pstfs	v3.1			154	Prefixed Store Floating-Point Single MLS:D-form
110101	I	.XXX			stfsu	P1			155	Store Floating-Point Single with Update D-form
000001 000// .//.. .... 11011.	I	..XX			pstxv	v3.1			591	Prefixed Store VSX Vector 8LS:D-form
110110	I	.XXX			stfd	P1			156	Store Floating-Point Double D-form
000001 100// .//.. .... 110110	I	..XX			pstfd	v3.1			156	Prefixed Store Floating-Point Double MLS:D-form
110111	I	.XXX			stfdu	P1			156	Store Floating-Point Double with Update D-form
111000	I	..XX			lq	v2.03			63	Load Quadword DQ-form
000001 000// .//.. .... 111000	I	..XX			plq	v3.1			63	Prefixed Load Quadword 8LS:D-form
000001 000// .//.. .... 111001	I	..XX			pld	v3.1			55	Prefixed Load Doubleword 8LS:D-form
111001 00	I	...X			lfdp	v2.05			158	Load Floating-Point Double Pair DS-form
111001 10	I	..XX			lxsd	v3.0			562	Load VSX Scalar Doubleword DS-form
111001 11	I	..XX			lxssp	v3.0			567	Load VSX Scalar Single-Precision DS-form
000001 000// .//.. .... 111010	I	..XX			plxvp	v3.1			603	Prefixed Load VSX Vector Paired 8LS:D-form
111010 00	I	..XX			ld	PPC			55	Load Doubleword DS-form
111010 01	I	..XX			ldu	PPC			55	Load Doubleword with Update DS-form
111010 10	I	..XX			lwa	PPC			54	Load Word Algebraic DS-form
111011 .... .0010 00010.	I	...X	DFP		dscli[.]	v2.05			233	DFP Shift Significand Left Immediate Z22-form
111011 .... .0011 00010.	I	...X	DFP		dscri[.]	v2.05			233	DFP Shift Significand Right Immediate Z22-form
111011 .../// .... .0110 00010/	I	...X	DFP		dtstdc	v2.05			210	DFP Test Data Class Z22-form
111011 .../// .... .0111 00010/	I	...X	DFP		dtstdg	v2.05			210	DFP Test Data Group Z22-form
111011 .... 00000 00010.	I	...X	DFP		dadd[.]	v2.05			204	DFP Add X-form
111011 .... 00001 00010.	I	...X	DFP		dmul[.]	v2.05			206	DFP Multiply X-form
111011 .../// 00100 00010/	I	...X	DFP		dcmpo	v2.05			209	DFP Compare Ordered X-form
111011 .../// 00101 00010/	I	...X	DFP		dtstex	v2.05			211	DFP Test Exponent X-form
111011 ..... ///// ..... 01000 00010.	I	...X	DFP		dctdp[.]	v2.05			225	DFP Convert To DFP Long X-form
111011 ..... ///// ..... 01001 00010.	I	...X	DFP		dctfix[.]	v2.05			228	DFP Convert To Fixed X-form
111011 ..... .///// ..... 01010 00010.	I	...X	DFP		ddedpd[.]	v2.05			230	DFP Decode DPD To BCD X-form
111011 ..... ///// ..... 01011 00010.	I	...X	DFP		dxex[.]	v2.05			231	DFP Extract Biased Exponent X-form
111011 .... 10000 00010.	I	...X	DFP		dsub[.]	v2.05			205	DFP Subtract X-form
111011 .... 10001 00010.	I	...X	DFP		ddiv[.]	v2.05			207	DFP Divide X-form
111011 .../// 10100 00010/	I	...X	DFP		dcmpu	v2.05			208	DFP Compare Unordered X-form
111011 .../// 10101 00010/	I	...X	DFP		dtstsf	v2.05			212	DFP Test Significance X-form
111011 ..... ///// ..... 11000 00010.	I	...X	DFP		drsp[.]	v2.05			226	DFP Round To DFP Short X-form
111011 ..... ///// ..... 11001 00010.	I	...X	DFP		dcffix[.]	v2.06			227	DFP Convert From Fixed X-form
111011 ..... .//// ..... 11010 00010.	I	...X	DFP		denbcd[.]	v2.05			230	DFP Encode BCD To DPD X-form
111011 .... 11011 00010.	I	...X	DFP		diex[.]	v2.05			231	DFP Insert Biased Exponent X-form
111011 ....000 00011.	I	...X	DFP		dqua[.]	v2.05			215	DFP Quantize Z23-form
111011 ....001 00011.	I	...X	DFP		drrnd[.]	v2.05			217	DFP Reround Z23-form

Table E.1: Power ISA Instruction Set Sorted by Opcode (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
111011 ....010 00011.	I	...X	DFP		dquai[.]	v2.05			214	DFP Quantize Immediate Z23-form
111011 ..... ////. ....011 00011.	I	...X	DFP		drintx[.]	v2.05			220	DFP Round To FP Integer With Inexact Z23-form
111011 ..... ////. ....111 00011.	I	...X	DFP		drintn[.]	v2.05			222	DFP Round To FP Integer Without Inexact Z23-form
111011 ...// 10101 00011/	I	...X	DFP		dtstsfi	v3.0			213	DFP Test Significance Immediate X-form
111011 ...// .... 00000 010../	I	....	MMA	MMA	xvi8ger4pp	v3.1			885	VSX Vector 8-bit Signed/Unsigned Integer GER (rank-4 update) Positive multiply, Positive accumulate XX3-form
000001 11100 1//// ..../ ///.. ..... 111011 ...// .... 00000 010../	I	....	MMA	MMA	pmxvi8ger4pp	v3.1			885	Prefixed Masked VSX Vector 8-bit Signed/Unsigned Integer GER (rank-4 update) Positive multiply, Positive accumulate MMIRR:XX3-form
111011 ...// .... 00010 010../	I	....	MMA	MMA	xvf16ger2pp	v3.1			894	VSX Vector 16-bit Floating-Point GER (rank-2 update) Positive multiply, Positive accumulate XX3-form
000001 11100 1//// ..//// ///.. ..... 111011 ...// .... 00010 010../	I	....	MMA	MMA	pmxvf16ger2pp	v3.1			894	Prefixed Masked VSX Vector 16-bit Floating-Point GER (rank-2 update) Positive multiply, Positive accumulate MMIRR:XX3-form
111011 ...// .... 00011 010../	I	....	MMA	MMA	xvf32gerpp	v3.1			899	VSX Vector 32-bit Floating-Point GER (rank-1 update) Positive multiply, Positive accumulate XX3-form
000001 11100 1//// ///// ///.. ..... 111011 ...// .... 00011 010../	I	....	MMA	MMA	pmxvf32gerpp	v3.1			899	Prefixed Masked VSX Vector 32-bit Floating-Point GER (rank-1 update) Positive multiply, Positive accumulate MMIRR:XX3-form
111011 ...// .... 00100 010../	I	....	MMA	MMA	xvi4ger8pp	v3.1			882	VSX Vector 4-bit Signed Integer GER (rank-8 update) Positive multiply, Positive accumulate XX3-form
000001 11100 1//// .... 111011 ...// .... 00100 010../	I	....	MMA	MMA	pmxvi4ger8pp	v3.1			882	Prefixed Masked VSX Vector 4-bit Signed Integer GER (rank-8 update) Positive multiply, Positive accumulate MMIRR:XX3-form
111011 ...// .... 00101 010../	I	....	MMA	MMA	xvi16ger2spp	v3.1			880	VSX Vector 16-bit Signed Integer GER (rank-2 update) with Saturation Positive multiply, Positive accumulate XX3-form
000001 11100 1//// ..//// ///.. ..... 111011 ...// .... 00101 010../	I	....	MMA	MMA	pmxvi16ger2spp	v3.1			880	Prefixed Masked VSX Vector 16-bit Signed Integer GER (rank-2 update) with Saturation Positive multiply, Positive accumulate MMIRR:XX3-form
111011 ...// .... 00110 010../	I	....	MMA	MMA	xvbf16ger2pp	v3.1			889	VSX Vector bfloat16 GER (rank-2 update) Positive multiply, Positive accumulate XX3-form
000001 11100 1//// ..//// ///.. ..... 111011 ...// .... 00110 010../	I	....	MMA	MMA	pmxvbf16ger2pp	v3.1			889	Prefixed Masked VSX Vector bfloat16 GER (rank-2 update) Positive multiply, Positive accumulate MMIRR:XX3-form
111011 ...// .... 00111 010../	I	....	MMA	MMA	xvf64gerpp	v3.1			903	VSX Vector 64-bit Floating-Point GER (rank-1 update) Positive multiply, Positive accumulate XX3-form
000001 11100 1//// ///// ///.. ..... 111011 ...// .... 00111 010../	I	....	MMA	MMA	pmxvf64gerpp	v3.1			903	Prefixed Masked VSX Vector 64-bit Floating-Point GER (rank-1 update) Positive multiply, Positive accumulate MMIRR:XX3-form
111011 ...// .... 01010 010../	I	....	MMA	MMA	xvf16ger2np	v3.1			894	VSX Vector 16-bit Floating-Point GER (rank-2 update) Negative multiply, Positive accumulate XX3-form
000001 11100 1//// ..//// ///.. ..... 111011 ...// .... 01010 010../	I	....	MMA	MMA	pmxvf16ger2np	v3.1			895	Prefixed Masked VSX Vector 16-bit Floating-Point GER (rank-2 update) Negative multiply, Positive accumulate MMIRR:XX3-form
111011 ...// .... 01011 010../	I	....	MMA	MMA	xvf32gernp	v3.1			899	VSX Vector 32-bit Floating-Point GER (rank-1 update) Negative multiply, Positive accumulate XX3-form

Table E.1: Power ISA Instruction Set Sorted by Opcode (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
000001 11100 1//// ///// ///.. ..... 111011 .../// .... 01011 010../	I	....	MMA	MMA	pmxvf32gernp	v3.1			900	Prefixed Masked VSX Vector 32-bit Floating-Point GER (rank-1 update) Negative multiply, Positive accumulate MMIRR:XX3-form
111011 .../// .... 01110 010../	I	....	MMA	MMA	xvbf16ger2np	v3.1			889	VSX Vector bfloat16 GER (rank-2 update) Negative multiply, Positive accumulate XX3-form
000001 11100 1//// ..//// ///.. ..... 111011 .../// .... 01110 010../	I	....	MMA	MMA	pmxvbf16ger2np	v3.1			890	Prefixed Masked VSX Vector bfloat16 GER (rank-2 update) Negative multiply, Positive accumulate MMIRR:XX3-form
111011 .../// .... 01111 010../	I	....	MMA	MMA	xvf64gernp	v3.1			903	VSX Vector 64-bit Floating-Point GER (rank-1 update) Negative multiply, Positive accumulate XX3-form
000001 11100 1//// ///// ///.. ..... 111011 .../// .... 01111 010../	I	....	MMA	MMA	pmxvf64gernp	v3.1			904	Prefixed Masked VSX Vector 64-bit Floating-Point GER (rank-1 update) Negative multiply, Positive accumulate MMIRR:XX3-form
111011 .../// .... 10010 010../	I	....	MMA	MMA	xvf16ger2pn	v3.1			894	VSX Vector 16-bit Floating-Point GER (rank-2 update) Positive multiply, Negative accumulate XX3-form
000001 11100 1//// ..//// ///.. ..... 111011 .../// .... 10010 010../	I	....	MMA	MMA	pmxvf16ger2pn	v3.1			894	Prefixed Masked VSX Vector 16-bit Floating-Point GER (rank-2 update) Positive multiply, Negative accumulate MMIRR:XX3-form
111011 .../// .... 10011 010../	I	....	MMA	MMA	xvf32gerpn	v3.1			899	VSX Vector 32-bit Floating-Point GER (rank-1 update) Positive multiply, Negative accumulate XX3-form
000001 11100 1//// ///// ///.. ..... 111011 .../// .... 10011 010../	I	....	MMA	MMA	pmxvf32gerpn	v3.1			899	Prefixed Masked VSX Vector 32-bit Floating-Point GER (rank-1 update) Positive multiply, Negative accumulate MMIRR:XX3-form
111011 .../// .... 10110 010../	I	....	MMA	MMA	xvbf16ger2pn	v3.1			889	VSX Vector bfloat16 GER (rank-2 update) Positive multiply, Negative accumulate XX3-form
000001 11100 1//// ..//// ///.. ..... 111011 .../// .... 10110 010../	I	....	MMA	MMA	pmxvbf16ger2pn	v3.1			889	Prefixed Masked VSX Vector bfloat16 GER (rank-2 update) Positive multiply, Negative accumulate MMIRR:XX3-form
111011 .../// .... 10111 010../	I	....	MMA	MMA	xvf64gerpn	v3.1			903	VSX Vector 64-bit Floating-Point GER (rank-1 update) Positive multiply, Negative accumulate XX3-form
000001 11100 1//// ///// ///.. ..... 111011 .../// .... 10111 010../	I	....	MMA	MMA	pmxvf64gerpn	v3.1			903	Prefixed Masked VSX Vector 64-bit Floating-Point GER (rank-1 update) Positive multiply, Negative accumulate MMIRR:XX3-form
111011 .../// .... 11010 010../	I	....	MMA	MMA	xvf16ger2nn	v3.1			894	VSX Vector 16-bit Floating-Point GER (rank-2 update) Negative multiply, Negative accumulate XX3-form
000001 11100 1//// ..//// ///.. ..... 111011 .../// .... 11010 010../	I	....	MMA	MMA	pmxvf16ger2nn	v3.1			895	Prefixed Masked VSX Vector 16-bit Floating-Point GER (rank-2 update) Negative multiply, Negative accumulate MMIRR:XX3-form
111011 .../// .... 11011 010../	I	....	MMA	MMA	xvf32gernn	v3.1			899	VSX Vector 32-bit Floating-Point GER (rank-1 update) Negative multiply, Negative accumulate XX3-form
000001 11100 1//// ///// ///.. ..... 111011 .../// .... 11011 010../	I	....	MMA	MMA	pmxvf32gernn	v3.1			900	Prefixed Masked VSX Vector 32-bit Floating-Point GER (rank-1 update) Negative multiply, Negative accumulate MMIRR:XX3-form
111011 .../// .... 11110 010../	I	....	MMA	MMA	xvbf16ger2nn	v3.1			889	VSX Vector bfloat16 GER (rank-2 update) Negative multiply, Negative accumulate XX3-form
000001 11100 1//// ..//// ///.. ..... 111011 .../// .... 11110 010../	I	....	MMA	MMA	pmxvbf16ger2nn	v3.1			890	Prefixed Masked VSX Vector bfloat16 GER (rank-2 update) Negative multiply, Negative accumulate MMIRR:XX3-form

Table E.1: Power ISA Instruction Set Sorted by Opcode (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
111011 ...// .... 11111 010../	I	....	MMA	MMA	xvf64gernn	v3.1			903	VSX Vector 64-bit Floating-Point GER (rank-1 update) Negative multiply, Negative accumulate XX3-form
000001 11100 1//// ///// ///.. ..... 111011 ...// .... 11111 010../	I	....	MMA	MMA	pmxvf64gernn	v3.1			904	Prefixed Masked VSX Vector 64-bit Floating-Point GER (rank-1 update) Negative multiply, Negative accumulate MMIRR:XX3-form
111011 ...// .... 00000 011../	I	....	MMA	MMA	xvi8ger4	v3.1			885	VSX Vector 8-bit Signed/Unsigned Integer GER (rank-4 update) XX3-form
000001 11100 1//// ....// ///.. ..... 111011 ...// .... 00000 011../	I	....	MMA	MMA	pmxvi8ger4	v3.1			885	Prefixed Masked VSX Vector 8-bit Signed/Unsigned Integer GER (rank-4 update) MMIRR:XX3-form
111011 ...// .... 00010 011../	I	....	MMA	MMA	xvf16ger2	v3.1			894	VSX Vector 16-bit Floating-Point GER (rank-2 update) XX3-form
000001 11100 1//// ..//// ///.. ..... 111011 ...// .... 00010 011../	I	....	MMA	MMA	pmxvf16ger2	v3.1			894	Prefixed Masked VSX Vector 16-bit Floating-Point GER (rank-2 update) MMIRR:XX3-form
111011 ...// .... 00011 011../	I	....	MMA	MMA	xvf32ger	v3.1			899	VSX Vector 32-bit Floating-Point GER (rank-1 update) XX3-form
000001 11100 1//// ///// ///.. ..... 111011 ...// .... 00011 011../	I	....	MMA	MMA	pmxvf32ger	v3.1			899	Prefixed Masked VSX Vector 32-bit Floating-Point GER (rank-1 update) MMIRR:XX3-form
111011 ...// .... 00100 011../	I	....	MMA	MMA	xvi4ger8	v3.1			882	VSX Vector 4-bit Signed Integer GER (rank-8 update) XX3-form
000001 11100 1//// .... 111011 ...// .... 00100 011../	I	....	MMA	MMA	pmxvi4ger8	v3.1			882	Prefixed Masked VSX Vector 4-bit Signed Integer GER (rank-8 update) MMIRR:XX3-form
111011 ...// .... 00101 011../	I	....	MMA	MMA	xvi16ger2s	v3.1			880	VSX Vector 16-bit Signed Integer GER (rank-2 update) with Saturation XX3-form
000001 11100 1//// ..//// ///.. ..... 111011 ...// .... 00101 011../	I	....	MMA	MMA	pmxvi16ger2s	v3.1			880	Prefixed Masked VSX Vector 16-bit Signed Integer GER (rank-2 update) with Saturation MMIRR:XX3-form
111011 ...// .... 00110 011../	I	....	MMA	MMA	xvbf16ger2	v3.1			889	VSX Vector bfloat16 GER (rank-2 update) XX3-form
000001 11100 1//// ..//// ///.. ..... 111011 ...// .... 00110 011../	I	....	MMA	MMA	pmxvbf16ger2	v3.1			889	Prefixed Masked VSX Vector bfloat16 GER (rank-2 update) MMIRR:XX3-form
111011 ...// .... 00111 011../	I	....	MMA	MMA	xvf64ger	v3.1			903	VSX Vector 64-bit Floating-Point GER (rank-1 update) XX3-form
000001 11100 1//// ///// ///.. ..... 111011 ...// .... 00111 011../	I	....	MMA	MMA	pmxvf64ger	v3.1			903	Prefixed Masked VSX Vector 64-bit Floating-Point GER (rank-1 update) MMIRR:XX3-form
111011 ...// .... 01001 011../	I	....	MMA	MMA	xvi16ger2	v3.1			878	VSX Vector 16-bit Signed Integer GER (rank-2 update) XX3-form
000001 11100 1//// ..//// ///.. ..... 111011 ...// .... 01001 011../	I	....	MMA	MMA	pmxvi16ger2	v3.1			878	Prefixed Masked VSX Vector 16-bit Signed Integer GER (rank-2 update) MMIRR:XX3-form
111011 ...// .... 01100 011../	I	....	MMA	MMA	xvi8ger4spp	v3.1			887	VSX Vector 8-bit Signed/Unsigned Integer GER (rank-4 update) with Saturation Positive multiply, Positive accumulate XX3-form
000001 11100 1//// ....// ///.. ..... 111011 ...// .... 01100 011../	I	....	MMA	MMA	pmxvi8ger4spp	v3.1			887	Prefixed Masked VSX Vector 8-bit Signed/Unsigned Integer GER (rank-4 update) with Saturation Positive multiply, Positive accumulate MMIRR:XX3-form
111011 ...// .... 01101 011../	I	....	MMA	MMA	xvi16ger2pp	v3.1			878	VSX Vector 16-bit Signed Integer GER (rank-2 update) Positive multiply, Positive accumulate XX3-form
000001 11100 1//// ..//// ///.. ..... 111011 ...// .... 01101 011../	I	....	MMA	MMA	pmxvi16ger2pp	v3.1			878	Prefixed Masked VSX Vector 16-bit Signed Integer GER (rank-2 update) Positive multiply, Positive accumulate MMIRR:XX3-form
111011 ..... ///// ..... 11010 01110.	I	.XXX			fcfids[.]	v2.06			176	Floating Convert with round Signed Doubleword to Single-Precision format X-form

Table E.1: Power ISA Instruction Set Sorted by Opcode (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
111011 ..... ///// ..... 11110 01110.	I	.XXX			fcfidus[.]	v2.06			176	Floating Convert with round Unsigned Doubleword to Single-Precision format X-form
111011 .... ///// 10010.	I	.XXX			fdivs[.]	PPC			163	Floating Divide Single A-form
111011 .... ///// 10100.	I	.XXX			fsubs[.]	PPC			162	Floating Subtract Single A-form
111011 .... ///// 10101.	I	.XXX			fadds[.]	PPC			162	Floating Add Single A-form
111011 ..... ///// ..... ///// 10110.	I	.XXX			fsqrts[.]	PPC			164	Floating Square Root Single A-form
111011 ..... ///// ..... ///// 11000.	I	.XXX			fres[.]	PPC			165	Floating Reciprocal Estimate Single A-form
111011 .... ///// ..... 11001.	I	.XXX			fmuls[.]	PPC			163	Floating Multiply Single A-form
111011 ..... ///// ..... ///// 11010.	I	.XXX			frsqrtes[.]	v2.02			166	Floating Reciprocal Square Root Estimate Single A-form
111011 .... 11100.	I	.XXX			fmsubs[.]	PPC			168	Floating Multiply-Subtract Single A-form
111011 .... 11101.	I	.XXX			fmadds[.]	PPC			168	Floating Multiply-Add Single A-form
111011 .... 11110.	I	.XXX			fnmsubs[.]	PPC			169	Floating Negative Multiply-Subtract Single A-form
111011 .... 11111.	I	.XXX			fnmadds[.]	PPC			169	Floating Negative Multiply-Add Single A-form
000001 000// .//.. .... 111100	I	..XX			pstq	v3.1			64	Prefixed Store Quadword 8LS:D-form
111100 .... 00000 000...	I	..XX			xsaddsp	v2.07			622	VSX Scalar Add Single-Precision XX3-form
111100 .... 00001 000...	I	..XX			xssubsp	v2.07			646	VSX Scalar Subtract Single-Precision XX3-form
111100 .... 00010 000...	I	..XX			xsmulsp	v2.07			634	VSX Scalar Multiply Single-Precision XX3-form
111100 .... 00011 000...	I	..XX			xsdivsp	v2.07			628	VSX Scalar Divide Single-Precision XX3-form
111100 .... 00100 000...	I	..XX			xsadddp	v2.06			615	VSX Scalar Add Double-Precision XX3-form
111100 .... 00101 000...	I	..XX			xssubdp	v2.06			642	VSX Scalar Subtract Double-Precision XX3-form
111100 .... 00110 000...	I	..XX			xsmuldp	v2.06			630	VSX Scalar Multiply Double-Precision XX3-form
111100 .... 00111 000...	I	..XX			xsdivdp	v2.06			624	VSX Scalar Divide Double-Precision XX3-form
111100 .... 01000 000...	I	..XX			xvaddsp	v2.06			694	VSX Vector Add Single-Precision XX3-form
111100 .... 01001 000...	I	..XX			xvsubsp	v2.06			708	VSX Vector Subtract Single-Precision XX3-form
111100 .... 01010 000...	I	..XX			xvmulsp	v2.06			702	VSX Vector Multiply Single-Precision XX3-form
111100 .... 01011 000...	I	..XX			xvdivsp	v2.06			698	VSX Vector Divide Single-Precision XX3-form
111100 .... 01100 000...	I	..XX			xvadddp	v2.06			691	VSX Vector Add Double-Precision XX3-form
111100 .... 01101 000...	I	..XX			xvsubdp	v2.06			706	VSX Vector Subtract Double-Precision XX3-form
111100 .... 01110 000...	I	..XX			xvmuldp	v2.06			700	VSX Vector Multiply Double-Precision XX3-form
111100 .... 01111 000...	I	..XX			xvdivdp	v2.06			696	VSX Vector Divide Double-Precision XX3-form
111100 .... 10000 000...	I	..XX			xsmaxcdp	v3.0			755	VSX Scalar Maximum Type-C Double-Precision XX3-form
111100 .... 10001 000...	I	..XX			xsmincdp	v3.0			763	VSX Scalar Minimum Type-C Double-Precision XX3-form
111100 .... 10010 000...	I	..XX			xsmaxjdp	v3.0			761	VSX Scalar Maximum Type-J Double-Precision XX3-form
111100 .... 10011 000...	I	..XX			xsminjdp	v3.0			769	VSX Scalar Minimum Type-J Double-Precision XX3-form
111100 .... 10100 000...	I	..XX			xsmaxdp	v2.06			759	VSX Scalar Maximum Double-Precision XX3-form
111100 .... 10101 000...	I	..XX			xsmindp	v2.06			767	VSX Scalar Minimum Double-Precision XX3-form
111100 .... 10110 000...	I	..XX			xscpsgndp	v2.06			608	VSX Scalar Copy Sign Double-Precision XX3-form
111100 .... 11000 000...	I	..XX			xvmaxsp	v2.06			779	VSX Vector Maximum Single-Precision XX3-form
111100 .... 11001 000...	I	..XX			xvminsp	v2.06			783	VSX Vector Minimum Single-Precision XX3-form
111100 .... 11010 000...	I	..XX			xvcpsgnsp	v2.06			612	VSX Vector Copy Sign Single-Precision XX3-form
111100 .... 11011 000...	I	..XX			xviexpsp	v3.0			871	VSX Vector Insert Exponent Single-Precision XX3-form
111100 .... 11100 000...	I	..XX			xvmaxdp	v2.06			777	VSX Vector Maximum Double-Precision XX3-form
111100 .... 11101 000...	I	..XX			xvmindp	v2.06			781	VSX Vector Minimum Double-Precision XX3-form
111100 .... 11110 000...	I	..XX			xvcpsgndp	v2.06			612	VSX Vector Copy Sign Double-Precision XX3-form
111100 .... 11111 000...	I	..XX			xviexpdp	v3.0			871	VSX Vector Insert Exponent Double-Precision XX3-form
111100 .... 00000 001...	I	..XX			xsmaddasp	v2.07			651	VSX Scalar Multiply-Add Type-A Single-Precision XX3-form

Table E.1: Power ISA Instruction Set Sorted by Opcode (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
111100 .... 00001 001...	I	..XX			xsmaddmsp	v2.07			651	VSX Scalar Multiply-Add Type-M Single-Precision XX3-form
111100 .... 00010 001...	I	..XX			xsmsubasp	v2.07			660	VSX Scalar Multiply-Subtract Type-A Single-Precision XX3-form
111100 .... 00011 001...	I	..XX			xsmsubmsp	v2.07			660	VSX Scalar Multiply-Subtract Type-M Single-Precision XX3-form
111100 .... 00100 001...	I	..XX			xsmaddadp	v2.06			648	VSX Scalar Multiply-Add Type-A Double-Precision XX3-form
111100 .... 00101 001...	I	..XX			xsmaddmdp	v2.06			648	VSX Scalar Multiply-Add Type-M Double-Precision XX3-form
111100 .... 00110 001...	I	..XX			xsmsubadp	v2.06			657	VSX Scalar Multiply-Subtract Type-A Double-Precision XX3-form
111100 .... 00111 001...	I	..XX			xsmsubmdp	v2.06			657	VSX Scalar Multiply-Subtract Type-M Double-Precision XX3-form
111100 .... 01000 001...	I	..XX			xvmaddasp	v2.06			713	VSX Vector Multiply-Add Type-A Single-Precision XX3-form
111100 .... 01001 001...	I	..XX			xvmaddmsp	v2.06			713	VSX Vector Multiply-Add Type-M Single-Precision XX3-form
111100 .... 01010 001...	I	..XX			xvmsubasp	v2.06			719	VSX Vector Multiply-Subtract Type-A Single-Precision XX3-form
111100 .... 01011 001...	I	..XX			xvmsubmsp	v2.06			719	VSX Vector Multiply-Subtract Type-M Single-Precision XX3-form
111100 .... 01100 001...	I	..XX			xvmaddadp	v2.06			710	VSX Vector Multiply-Add Type-A Double-Precision XX3-form
111100 .... 01101 001...	I	..XX			xvmaddmdp	v2.06			710	VSX Vector Multiply-Add Type-M Double-Precision XX3-form
111100 .... 01110 001...	I	..XX			xvmsubadp	v2.06			716	VSX Vector Multiply-Subtract Type-A Double-Precision XX3-form
111100 .... 01111 001...	I	..XX			xvmsubmdp	v2.06			716	VSX Vector Multiply-Subtract Type-M Double-Precision XX3-form
111100 .... 10000 001...	I	..XX			xsnmaddasp	v2.07			670	VSX Scalar Negative Multiply-Add Type-A Single-Precision XX3-form
111100 .... 10001 001...	I	..XX			xsnmaddmsp	v2.07			670	VSX Scalar Negative Multiply-Add Type-M Single-Precision XX3-form
111100 .... 10010 001...	I	..XX			xsnmsubasp	v2.07			679	VSX Scalar Negative Multiply-Subtract Type-A Single-Precision XX3-form
111100 .... 10011 001...	I	..XX			xsnmsubmsp	v2.07			679	VSX Scalar Negative Multiply-Subtract Type-M Single-Precision XX3-form
111100 .... 10100 001...	I	..XX			xsnmaddadp	v2.06			666	VSX Scalar Negative Multiply-Add Type-A Double-Precision XX3-form
111100 .... 10101 001...	I	..XX			xsnmaddmdp	v2.06			666	VSX Scalar Negative Multiply-Add Type-M Double-Precision XX3-form
111100 .... 10110 001...	I	..XX			xsnmsubadp	v2.06			676	VSX Scalar Negative Multiply-Subtract Type-A Double-Precision XX3-form
111100 .... 10111 001...	I	..XX			xsnmsubmdp	v2.06			676	VSX Scalar Negative Multiply-Subtract Type-M Double-Precision XX3-form
111100 .... 11000 001...	I	..XX			xvnmaddasp	v2.06			726	VSX Vector Negative Multiply-Add Type-A Single-Precision XX3-form

Table E.1: Power ISA Instruction Set Sorted by Opcode (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
111100 .... 11001 001...	I	..XX			xvnmaddmsp	v2.06			726	VSX Vector Negative Multiply-Add Type-M Single-Precision XX3-form
111100 .... 11010 001...	I	..XX			xvnmsubasp	v2.06			732	VSX Vector Negative Multiply-Subtract Type-A Single-Precision XX3-form
111100 .... 11011 001...	I	..XX			xvnmsubmsp	v2.06			732	VSX Vector Negative Multiply-Subtract Type-M Single-Precision XX3-form
111100 .... 11100 001...	I	..XX			xvnmaddadp	v2.06			722	VSX Vector Negative Multiply-Add Type-A Double-Precision XX3-form
111100 .... 11101 001...	I	..XX			xvnmaddmdp	v2.06			722	VSX Vector Negative Multiply-Add Type-M Double-Precision XX3-form
111100 .... 11110 001...	I	..XX			xvnmsubadp	v2.06			729	VSX Vector Negative Multiply-Subtract Type-A Double-Precision XX3-form
111100 .... 11111 001...	I	..XX			xvnmsubmdp	v2.06			729	VSX Vector Negative Multiply-Subtract Type-M Double-Precision XX3-form
111100 .... 0..00 010...	I	..XX			xxsldwi	v2.06			925	VSX Vector Shift Left Double by Word Immediate XX3-form
111100 .... 0..01 010...	I	..XX			xxpermdi	v2.06			922	VSX Vector Permute Doubleword Immediate XX3-form
111100 .... 00010 010...	I	..XX			xxmrghw	v2.06			918	VSX Vector Merge High Word XX3-form
111100 .... 00011 010...	I	..XX			xxperm	v3.0			923	VSX Vector Permute XX3-form
111100 .... 00110 010...	I	..XX			xxmrglw	v2.06			918	VSX Vector Merge Low Word XX3-form
111100 .... 00111 010...	I	..XX			xxpermr	v3.0			923	VSX Vector Permute Right-indexed XX3-form
111100 .... 10000 010...	I	..XX			xxland	v2.06			907	VSX Vector Logical AND XX3-form
111100 .... 10001 010...	I	..XX			xxlandc	v2.06			907	VSX Vector Logical AND with Complement XX3-form
111100 .... 10010 010...	I	..XX			xxlor	v2.06			908	VSX Vector Logical OR XX3-form
111100 .... 10011 010...	I	..XX			xxlxor	v2.06			908	VSX Vector Logical XOR XX3-form
111100 .... 10100 010...	I	..XX			xxlnor	v2.06			908	VSX Vector Logical NOR XX3-form
111100 .... 10101 010...	I	..XX			xxlorc	v2.07			908	VSX Vector Logical OR with Complement XX3-form
111100 .... 10110 010...	I	..XX			xxlnand	v2.07			907	VSX Vector Logical NAND XX3-form
111100 .... 10111 010...	I	..XX			xxleqv	v2.07			907	VSX Vector Logical Equivalence XX3-form
111100 ..... ///.. ..... 01010 0100..	I	..XX			xxspltw	v2.06			921	VSX Vector Splat Word XX2-form
111100 ..... 00... ..... 01011 01000.	I	..XX			xxspltib	v3.0			919	VSX Vector Splat Immediate Byte X-form
111100 ..... 11111 ..... 01011 01000.	I	..XX			lxvkq	v3.1			935	Load VSX Vector Special Value Quadword X-form
111100 ..... /.... ..... 01010 0101..	I	..XX			xxextractuw	v3.0			917	VSX Vector Extract Unsigned Word XX2-form
111100 ..... /.... ..... 01011 0101..	I	..XX			xxinsertw	v3.0			917	VSX Vector Insert Word XX2-form
111100 .... .1000 011...	I	..XX			xvcmpeqsp[.]	v2.06			772	VSX Vector Compare Equal To Single-Precision XX3-form
111100 .... .1001 011...	I	..XX			xvcmpgtsp[.]	v2.06			776	VSX Vector Compare Greater Than Single-Precision XX3-form
111100 .... .1010 011...	I	..XX			xvcmpgesp[.]	v2.06			774	VSX Vector Compare Greater Than or Equal To Single-Precision XX3-form
111100 .... .1100 011...	I	..XX			xvcmpeqdp[.]	v2.06			771	VSX Vector Compare Equal To Double-Precision XX3-form
111100 .... .1101 011...	I	..XX			xvcmpgtdp[.]	v2.06			775	VSX Vector Compare Greater Than Double-Precision XX3-form
111100 .... .1110 011...	I	..XX			xvcmpgedp[.]	v2.06			773	VSX Vector Compare Greater Than or Equal To Double-Precision XX3-form
111100 .... 00000 011...	I	..XX			xscmpeqdp	v3.0			749	VSX Scalar Compare Equal Double-Precision XX3-form
111100 .... 00001 011...	I	..XX			xscmpgtdp	v3.0			753	VSX Scalar Compare Greater Than Double-Precision XX3-form
111100 .... 00010 011...	I	..XX			xscmpgedp	v3.0			751	VSX Scalar Compare Greater Than or Equal Double-Precision XX3-form

Table E.1: Power ISA Instruction Set Sorted by Opcode (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
111100 ...// .... 00100 011../	I	..XX			xscmpudp	v2.06			746	VSX Scalar Compare Unordered Double-Precision XX3-form
111100 ...// .... 00101 011../	I	..XX			xscmpodp	v2.06			743	VSX Scalar Compare Ordered Double-Precision XX3-form
111100 ...// .... 00111 011../	I	..XX			xscmpexpdp	v3.0			862	VSX Scalar Compare Exponents Double-Precision XX3-form
111100 ..... ///// ..... 00100 1000..	I	..XX			xscvdpuxws	v2.06			822	VSX Scalar Convert with round to zero Double-Precision to Unsigned Word format XX2-form
111100 ..... ///// ..... 00101 1000..	I	..XX			xscvdpsxws	v2.06			818	VSX Scalar Convert with round to zero Double-Precision to Signed Word format XX2-form
111100 ..... ///// ..... 01000 1000..	I	..XX			xvcvspuxws	v2.06			850	VSX Vector Convert with round to zero Single-Precision to Unsigned Word format XX2-form
111100 ..... ///// ..... 01001 1000..	I	..XX			xvcvspsxws	v2.06			846	VSX Vector Convert with round to zero Single-Precision to Signed Word format XX2-form
111100 ..... ///// ..... 01010 1000..	I	..XX			xvcvuxwsp	v2.06			861	VSX Vector Convert with round Unsigned Word to Single-Precision format XX2-form
111100 ..... ///// ..... 01011 1000..	I	..XX			xvcvsxwsp	v2.06			861	VSX Vector Convert with round Signed Word to Single-Precision format XX2-form
111100 ..... ///// ..... 01100 1000..	I	..XX			xvcvdpuxws	v2.06			842	VSX Vector Convert with round to zero Double-Precision to Unsigned Word format XX2-form
111100 ..... ///// ..... 01101 1000..	I	..XX			xvcvdpsxws	v2.06			838	VSX Vector Convert with round to zero Double-Precision to Signed Word format XX2-form
111100 ..... ///// ..... 01110 1000..	I	..XX			xvcvuxwdp	v2.06			858	VSX Vector Convert Unsigned Word to Double-Precision format XX2-form
111100 ..... ///// ..... 01111 1000..	I	..XX			xvcvsxwdp	v2.06			858	VSX Vector Convert Signed Word to Double-Precision format XX2-form
111100 ..... ///// ..... 10010 1000..	I	..XX			xscvuxdsp	v2.07			855	VSX Scalar Convert with round Unsigned Doubleword to Single-Precision format XX2-form
111100 ..... ///// ..... 10011 1000..	I	..XX			xscvsxdsp	v2.07			855	VSX Scalar Convert with round Signed Doubleword to Single-Precision format XX2-form
111100 ..... ///// ..... 10100 1000..	I	..XX			xscvdpuxds	v2.06			820	VSX Scalar Convert with round to zero Double-Precision to Unsigned Doubleword format XX2-form
111100 ..... ///// ..... 10101 1000..	I	..XX			xscvdpsxds	v2.06			816	VSX Scalar Convert with round to zero Double-Precision to Signed Doubleword format XX2-form
111100 ..... ///// ..... 10110 1000..	I	..XX			xscvuxddp	v2.06			854	VSX Scalar Convert with round Unsigned Doubleword to Double-Precision format XX2-form
111100 ..... ///// ..... 10111 1000..	I	..XX			xscvsxddp	v2.06			854	VSX Scalar Convert with round Signed Doubleword to Double-Precision format XX2-form
111100 ..... ///// ..... 11000 1000..	I	..XX			xvcvspuxds	v2.06			848	VSX Vector Convert with round to zero Single-Precision to Unsigned Doubleword format XX2-form
111100 ..... ///// ..... 11001 1000..	I	..XX			xvcvspsxds	v2.06			844	VSX Vector Convert with round to zero Single-Precision to Signed Doubleword format XX2-form
111100 ..... ///// ..... 11010 1000..	I	..XX			xvcvuxdsp	v2.06			859	VSX Vector Convert with round Unsigned Doubleword to Single-Precision format XX2-form
111100 ..... ///// ..... 11011 1000..	I	..XX			xvcvsxdsp	v2.06			859	VSX Vector Convert with round Signed Doubleword to Single-Precision format XX2-form
111100 ..... ///// ..... 11100 1000..	I	..XX			xvcvdpuxds	v2.06			840	VSX Vector Convert with round to zero Double-Precision to Unsigned Doubleword format XX2-form
111100 ..... ///// ..... 11101 1000..	I	..XX			xvcvdpsxds	v2.06			836	VSX Vector Convert with round to zero Double-Precision to Signed Doubleword format XX2-form

Table E.1: Power ISA Instruction Set Sorted by Opcode (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
111100 ..... ///// ..... 11110 1000..	I	..XX			xvcvuxddp	v2.06			857	VSX Vector Convert with round Unsigned Doubleword to Double-Precision format XX2-form
111100 ..... ///// ..... 11111 1000..	I	..XX			xvcvsxddp	v2.06			857	VSX Vector Convert with round Signed Doubleword to Double-Precision format XX2-form
111100 ..... ///// ..... 00100 1001..	I	..XX			xsrdpi	v2.06			801	VSX Scalar Round to Double-Precision Integer using round to Nearest Away XX2-form
111100 ..... ///// ..... 00101 1001..	I	..XX			xsrdpiz	v2.06			805	VSX Scalar Round to Double-Precision Integer using round toward Zero XX2-form
111100 ..... ///// ..... 00110 1001..	I	..XX			xsrdpip	v2.06			804	VSX Scalar Round to Double-Precision Integer using round toward +Infinity XX2-form
111100 ..... ///// ..... 00111 1001..	I	..XX			xsrdpim	v2.06			803	VSX Scalar Round to Double-Precision Integer using round toward -Infinity XX2-form
111100 ..... ///// ..... 01000 1001..	I	..XX			xvrspi	v2.06			812	VSX Vector Round to Single-Precision Integer using round to Nearest Away XX2-form
111100 ..... ///// ..... 01001 1001..	I	..XX			xvrspiz	v2.06			815	VSX Vector Round to Single-Precision Integer using round toward Zero XX2-form
111100 ..... ///// ..... 01010 1001..	I	..XX			xvrspip	v2.06			814	VSX Vector Round to Single-Precision Integer using round toward +Infinity XX2-form
111100 ..... ///// ..... 01011 1001..	I	..XX			xvrspim	v2.06			814	VSX Vector Round to Single-Precision Integer using round toward -Infinity XX2-form
111100 ..... ///// ..... 01100 1001..	I	..XX			xvrdpi	v2.06			808	VSX Vector Round to Double-Precision Integer using round to Nearest Away XX2-form
111100 ..... ///// ..... 01101 1001..	I	..XX			xvrdpiz	v2.06			811	VSX Vector Round to Double-Precision Integer using round toward Zero XX2-form
111100 ..... ///// ..... 01110 1001..	I	..XX			xvrdpip	v2.06			810	VSX Vector Round to Double-Precision Integer using round toward +Infinity XX2-form
111100 ..... ///// ..... 01111 1001..	I	..XX			xvrdpim	v2.06			810	VSX Vector Round to Double-Precision Integer using round toward -Infinity XX2-form
111100 ..... ///// ..... 10000 1001..	I	..XX			xscvdpsp	v2.06			789	VSX Scalar Convert with round Double-Precision to Single-Precision format XX2-form
111100 ..... ///// ..... 10001 1001..	I	..XX			xsrsp	v2.07			787	VSX Scalar Round to Single-Precision XX2-form
111100 ..... ///// ..... 10100 1001..	I	..XX			xscvspdp	v2.06			797	VSX Scalar Convert Single-Precision to Double-Precision format XX2-form
111100 ..... ///// ..... 10101 1001..	I	..XX			xsabsdp	v2.06			607	VSX Scalar Absolute Double-Precision XX2-form
111100 ..... ///// ..... 10110 1001..	I	..XX			xsnabsdp	v2.06			609	VSX Scalar Negative Absolute Double-Precision XX2-form
111100 ..... ///// ..... 10111 1001..	I	..XX			xsnegdp	v2.06			610	VSX Scalar Negate Double-Precision XX2-form
111100 ..... ///// ..... 11000 1001..	I	..XX			xvcvdpsp	v2.06			793	VSX Vector Convert with round Double-Precision to Single-Precision format XX2-form
111100 ..... ///// ..... 11001 1001..	I	..XX			xvabssp	v2.06			611	VSX Vector Absolute Single-Precision XX2-form
111100 ..... ///// ..... 11010 1001..	I	..XX			xvnabssp	v2.06			613	VSX Vector Negative Absolute Single-Precision XX2-form
111100 ..... ///// ..... 11011 1001..	I	..XX			xvnegsp	v2.06			614	VSX Vector Negate Single-Precision XX2-form
111100 ..... ///// ..... 11100 1001..	I	..XX			xvcvspdp	v2.06			800	VSX Vector Convert Single-Precision to Double-Precision format XX2-form
111100 ..... ///// ..... 11101 1001..	I	..XX			xvabsdp	v2.06			611	VSX Vector Absolute Double-Precision XX2-form
111100 ..... ///// ..... 11110 1001..	I	..XX			xvnabsdp	v2.06			613	VSX Vector Negative Absolute Double-Precision XX2-form
111100 ..... ///// ..... 11111 1001..	I	..XX			xvnegdp	v2.06			614	VSX Vector Negate Double-Precision XX2-form
111100 .../// .... 00111 101../	I	..XX			xstdivdp	v2.06			689	VSX Scalar Test for software Divide Double-Precision XX3-form

Table E.1: Power ISA Instruction Set Sorted by Opcode (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
111100 ...// .... 01011 101../	I	..XX			xvtdivsp	v2.06			740	VSX Vector Test for software Divide Single-Precision XX3-form
111100 ...// .... 01111 101../	I	..XX			xvtdivdp	v2.06			739	VSX Vector Test for software Divide Double-Precision XX3-form
111100 .... 1101. 101...	I	..XX			xvtstdcsp	v3.0			873	VSX Vector Test Data Class Single-Precision XX2-form
111100 .... 1111. 101...	I	..XX			xvtstdcdp	v3.0			872	VSX Vector Test Data Class Double-Precision XX2-form
111100 ..... ///// ..... 00000 1010..	I	..XX			xsrsqrtesp	v2.07			688	VSX Scalar Reciprocal Square Root Estimate Single-Precision XX2-form
111100 ..... ///// ..... 00001 1010..	I	..XX			xsresp	v2.07			686	VSX Scalar Reciprocal Estimate Single-Precision XX2-form
111100 ..... ///// ..... 00100 1010..	I	..XX			xsrsqrtedp	v2.06			687	VSX Scalar Reciprocal Square Root Estimate Double-Precision XX2-form
111100 ..... ///// ..... 00101 1010..	I	..XX			xsredp	v2.06			685	VSX Scalar Reciprocal Estimate Double-Precision XX2-form
111100 ...// ///// 00110 1010./	I	..XX			xstsqrtdp	v2.06			690	VSX Scalar Test for software Square Root Double-Precision XX2-form
111100 ..... ///// ..... 01000 1010..	I	..XX			xvrsqrtesp	v2.06			738	VSX Vector Reciprocal Square Root Estimate Single-Precision XX2-form
111100 ..... ///// ..... 01001 1010..	I	..XX			xvresp	v2.06			736	VSX Vector Reciprocal Estimate Single-Precision XX2-form
111100 ...// ///// 01010 1010./	I	..XX			xvtsqrtsp	v2.06			742	VSX Vector Test for software Square Root Single-Precision XX2-form
111100 ..... ///// ..... 01100 1010..	I	..XX			xvrsqrtedp	v2.06			737	VSX Vector Reciprocal Square Root Estimate Double-Precision XX2-form
111100 ..... ///// ..... 01101 1010..	I	..XX			xvredp	v2.06			735	VSX Vector Reciprocal Estimate Double-Precision XX2-form
111100 ...// ///// 01110 1010./	I	..XX			xvtsqrtdp	v2.06			741	VSX Vector Test for software Square Root Double-Precision XX2-form
111100 .... 10010 1010./	I	..XX			xststdcsp	v3.0			868	VSX Scalar Test Data Class Single-Precision XX2-form
111100 .... 10110 1010./	I	..XX			xststdcdp	v3.0			866	VSX Scalar Test Data Class Double-Precision XX2-form
111100 .... 11100 10100.	I	..XX			xxgenpcvbm	v3.1			926	VSX Vector Generate PCV from Byte Mask X-form
111100 .... 11101 10100.	I	..XX			xxgenpcvwm	v3.1			933	VSX Vector Generate PCV from Word Mask X-form
111100 .... 11100 10101.	I	..XX			xxgenpcvhm	v3.1			931	VSX Vector Generate PCV from Halfword Mask X-form
111100 .... 11101 10101.	I	..XX			xxgenpcvdm	v3.1			928	VSX Vector Generate PCV from Doubleword Mask X-form
111100 ..... ///// ..... 00000 1011..	I	..XX			xssqrtsp	v2.07			640	VSX Scalar Square Root Single-Precision XX2-form
111100 ..... ///// ..... 00100 1011..	I	..XX			xssqrtdp	v2.06			636	VSX Scalar Square Root Double-Precision XX2-form
111100 ..... ///// ..... 00110 1011..	I	..XX			xsrdpic	v2.06			802	VSX Scalar Round to Double-Precision Integer exact using Current rounding mode XX2-form
111100 ..... ///// ..... 01000 1011..	I	..XX			xvsqrtsp	v2.06			705	VSX Vector Square Root Single-Precision XX2-form
111100 ..... ///// ..... 01010 1011..	I	..XX			xvrspic	v2.06			813	VSX Vector Round to Single-Precision Integer Exact using Current rounding mode XX2-form
111100 ..... ///// ..... 01100 1011..	I	..XX			xvsqrtdp	v2.06			704	VSX Vector Square Root Double-Precision XX2-form
111100 ..... ///// ..... 01110 1011..	I	..XX			xvrdpic	v2.06			809	VSX Vector Round to Double-Precision Integer Exact using Current rounding mode XX2-form
111100 ..... ///// ..... 10000 1011..	I	..XX			xscvdpspn	v2.07			790	VSX Scalar Convert Scalar Single-Precision to Vector Single-Precision format Non-signalling XX2-form
111100 ..... ///// ..... 10100 1011..	I	..XX			xscvspdpn	v2.07			798	VSX Scalar Convert Single-Precision to Double-Precision format Non-signalling XX2-form
111100 ..... 00000 ..... 10101 1011./	I	..XX			xsxexpdp	v3.0			869	VSX Scalar Extract Exponent Double-Precision XX2-form

Table E.1: Power ISA Instruction Set Sorted by Opcode (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
111100 ..... 00001 ..... 10101 1011./	I	..XX			xsxsigdp	v3.0			870	VSX Scalar Extract Significand Double-Precision XX2-form
111100 ..... 10000 ..... 10101 1011..	I	..XX			xscvhpdp	v3.0			796	VSX Scalar Convert Half-Precision to Double-Precision format XX2-form
111100 ..... 10001 ..... 10101 1011..	I	..XX			xscvdphp	v3.0			788	VSX Scalar Convert with round Double-Precision to Half-Precision format XX2-form
111100 ..... 00000 ..... 11101 1011..	I	..XX			xvxexpdp	v3.0			874	VSX Vector Extract Exponent Double-Precision XX2-form
111100 ..... 00001 ..... 11101 1011..	I	..XX			xvxsigdp	v3.0			875	VSX Vector Extract Significand Double-Precision XX2-form
111100 ...// 00010 ..... 11101 1011./	I	..XX			xvtlsbb	v3.1			936	VSX Vector Test Least-Significant Bit by Byte XX2-form
111100 ..... 00111 ..... 11101 1011..	I	..XX			xxbrh	v3.0			915	VSX Vector Byte-Reverse Halfword XX2-form
111100 ..... 01000 ..... 11101 1011..	I	..XX			xvxexpsp	v3.0			874	VSX Vector Extract Exponent Single-Precision XX2-form
111100 ..... 01001 ..... 11101 1011..	I	..XX			xvxsigsp	v3.0			875	VSX Vector Extract Significand Single-Precision XX2-form
111100 ..... 01111 ..... 11101 1011..	I	..XX			xxbrw	v3.0			916	VSX Vector Byte-Reverse Word XX2-form
111100 ..... 10000 ..... 11101 1011..	I	..XX			xvcvbf16spn	v3.1			800	VSX Vector Convert bfloat16 to Single-Precision format Non-signaling XX2-form
111100 ..... 10001 ..... 11101 1011..	I	..XX			xvcvspbf16	v3.1			792	VSX Vector Convert with round Single-Precision to bfloat16 format XX2-form
111100 ..... 10111 ..... 11101 1011..	I	..XX			xxbrd	v3.0			914	VSX Vector Byte-Reverse Doubleword XX2-form
111100 ..... 11000 ..... 11101 1011..	I	..XX			xvcvhpsp	v3.0			799	VSX Vector Convert Half-Precision to Single-Precision format XX2-form
111100 ..... 11001 ..... 11101 1011..	I	..XX			xvcvsphp	v3.0			794	VSX Vector Convert with round Single-Precision to Half-Precision format XX2-form
111100 ..... 11111 ..... 11101 1011..	I	..XX			xxbrq	v3.0			915	VSX Vector Byte-Reverse Quadword XX2-form
111100 .... 11100 10110.	I	..XX			xsiexpdp	v3.0			864	VSX Scalar Insert Exponent Double-Precision X-form
111100 .... 11....	I	..XX			xxsel	v2.06			909	VSX Vector Select XX4-form
000001 000// .//.. .... 111101	I	..XX			pstd	v3.1			60	Prefixed Store Doubleword 8LS:D-form
111101 00	I	...X			stfdp	v2.05			159	Store Floating-Point Double Pair DS-form
111101 10	I	..XX			stxsd	v3.0			569	Store VSX Scalar Doubleword DS-form
111101 11	I	..XX			stxssp	v3.0			573	Store VSX Scalar Single-Precision DS-form
111101 001	I	..XX			lxv	v3.0			575	Load VSX Vector DQ-form
111101 101	I	..XX			stxv	v3.0			591	Store VSX Vector DQ-form
000001 000// .//.. .... 111110	I	..XX			pstxvp	v3.1			605	Prefixed Store VSX Vector Paired 8LS:D-form
111110 00	I	..XX			std	PPC			60	Store Doubleword DS-form
111110 01	I	..XX			stdu	PPC			61	Store Doubleword with Update DS-form
111110 10	I	..XX			stq	v2.03			64	Store Quadword DS-form
111111 ...// 00000 00000/	I	.XXX			fcmpu	P1			179	Floating Compare Unordered X-form
111111 ...// 00001 00000/	I	.XXX			fcmpo	P1			179	Floating Compare Ordered X-form
111111 ...// ...// ///// 00010 00000/	I	.XXX			mcrfs	P1			184	Move to Condition Register from FPSCR X-form
111111 ...// 00100 00000/	I	.XXX			ftdiv	v2.06			167	Floating Test for software Divide X-form
111111 ...// ///// 00101 00000/	I	.XXX			ftsqrt	v2.06			167	Floating Test for software Square Root X-form
111111 .... .0010 00010.	I	...X	DFP		dscliq[.]	v2.05			233	DFP Shift Significand Left Immediate Quad Z22-form
111111 .... .0011 00010.	I	...X	DFP		dscriq[.]	v2.05			233	DFP Shift Significand Right Immediate Quad Z22-form
111111 ...// .... .0110 00010/	I	...X	DFP		dtstdcq	v2.05			210	DFP Test Data Class Quad Z22-form
111111 ...// .... .0111 00010/	I	...X	DFP		dtstdgq	v2.05			210	DFP Test Data Group Quad Z22-form
111111 .... 00000 00010.	I	...X	DFP		daddq[.]	v2.05			204	DFP Add Quad X-form
111111 .... 00001 00010.	I	...X	DFP		dmulq[.]	v2.05			206	DFP Multiply Quad X-form

Table E.1: Power ISA Instruction Set Sorted by Opcode (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
111111 ...// 00100 00010/	I	...X	DFP		dcmpoq	v2.05			209	DFP Compare Ordered Quad X-form
111111 ...// 00101 00010/	I	...X	DFP		dtstexq	v2.05			211	DFP Test Exponent Quad X-form
111111 ..... ///// ..... 01000 00010.	I	...X	DFP		dctqpq[.]	v2.05			225	DFP Convert To DFP Extended X-form
111111 ..... ///// ..... 01001 00010.	I	...X	DFP		dctfixq[.]	v2.05			228	DFP Convert To Fixed Quad X-form
111111 ....//// ..... 01010 00010.	I	...X	DFP		ddedpdq[.]	v2.05			230	DFP Decode DPD To BCD Quad X-form
111111 ..... ///// ..... 01011 00010.	I	...X	DFP		dxexq[.]	v2.05			231	DFP Extract Biased Exponent Quad X-form
111111 .... 10000 00010.	I	...X	DFP		dsubq[.]	v2.05			205	DFP Subtract Quad X-form
111111 .... 10001 00010.	I	...X	DFP		ddivq[.]	v2.05			207	DFP Divide Quad X-form
111111 ...// 10100 00010/	I	...X	DFP		dcmpuq	v2.05			208	DFP Compare Unordered Quad X-form
111111 ...// 10101 00010/	I	...X	DFP		dtstsfq	v2.05			212	DFP Test Significance Quad X-form
111111 ..... ///// ..... 11000 00010.	I	...X	DFP		drdpq[.]	v2.05			226	DFP Round To DFP Long X-form
111111 ..... ///// ..... 11001 00010.	I	...X	DFP		dcffixq[.]	v2.05			227	DFP Convert From Fixed Quad X-form
111111 ..... ./// ..... 11010 00010.	I	...X	DFP		denbcdq[.]	v2.05			230	DFP Encode BCD To DPD Quad X-form
111111 .... 11011 00010.	I	...X	DFP		diexq[.]	v2.05			231	DFP Insert Biased Exponent Quad X-form
111111 ..... 00000 ..... 11111 00010/	I	...X	DFP		dcffixqq	v3.1			227	DFP Convert From Fixed Quadword Quad X-form
111111 ..... 00001 ..... 11111 00010/	I	...X	DFP		dctfixqq	v3.1			228	DFP Convert To Fixed Quadword Quad X-form
111111 ....000 00011.	I	...X	DFP		dquaq[.]	v2.05			215	DFP Quantize Quad Z23-form
111111 ....001 00011.	I	...X	DFP		drrndq[.]	v2.05			217	DFP Reround Quad Z23-form
111111 ....010 00011.	I	...X	DFP		dquaiq[.]	v2.05			214	DFP Quantize Immediate Quad Z23-form
111111 ..... ////. ....011 00011.	I	...X	DFP		drintxq[.]	v2.05			220	DFP Round To FP Integer With Inexact Quad Z23-form
111111 ..... ////. ....111 00011.	I	...X	DFP		drintnq[.]	v2.05			222	DFP Round To FP Integer Without Inexact Quad Z23-form
111111 ...// 10101 00011/	I	...X	DFP		dtstsfiq	v3.0			213	DFP Test Significance Immediate Quad X-form
111111 .... 00000 00100.	I	...X	BFP128		xsaddqp[o]	v3.0			620	VSX Scalar Add Quad-Precision [using round to Odd] X-form
111111 .... 00001 00100.	I	...X	BFP128		xsmulqp[o]	v3.0			632	VSX Scalar Multiply Quad-Precision [using round to Odd] X-form
111111 .... 00010 00100/	I	..XX			xscmpeqqp	v3.1			750	VSX Scalar Compare Equal Quad-Precision X-form
111111 .... 00011 00100/	I	...X	BFP128		xscpsgnqp	v3.0			608	VSX Scalar Copy Sign Quad-Precision X-form
111111 ...// 00100 00100/	I	...X	BFP128		xscmpoqp	v3.0			745	VSX Scalar Compare Ordered Quad-Precision X-form
111111 ...// 00101 00100/	I	...X	BFP128		xscmpexpqp	v3.0			863	VSX Scalar Compare Exponents Quad-Precision X-form
111111 .... 00110 00100/	I	..XX			xscmpgeqp	v3.1			752	VSX Scalar Compare Greater Than or Equal Quad-Precision X-form
111111 .... 00111 00100/	I	..XX			xscmpgtqp	v3.1			754	VSX Scalar Compare Greater Than Quad-Precision X-form
111111 .... 01100 00100.	I	...X	BFP128		xsmaddqp[o]	v3.0			654	VSX Scalar Multiply-Add Quad-Precision [using round to Odd] X-form
111111 .... 01101 00100.	I	...X	BFP128		xsmsubqp[o]	v3.0			663	VSX Scalar Multiply-Subtract Quad-Precision [using round to Odd] X-form
111111 .... 01110 00100.	I	...X	BFP128		xsnmaddqp[o]	v3.0			673	VSX Scalar Negative Multiply-Add Quad-Precision [using round to Odd] X-form
111111 .... 01111 00100.	I	...X	BFP128		xsnmsubqp[o]	v3.0			682	VSX Scalar Negative Multiply-Subtract Quad-Precision [using round to Odd] X-form
111111 .... 10000 00100.	I	...X	BFP128		xssubqp[o]	v3.0			644	VSX Scalar Subtract Quad-Precision [using round to Odd] X-form
111111 .... 10001 00100.	I	...X	BFP128		xsdivqp[o]	v3.0			626	VSX Scalar Divide Quad-Precision [using round to Odd] X-form
111111 ...// 10100 00100/	I	...X	BFP128		xscmpuqp	v3.0			748	VSX Scalar Compare Unordered Quad-Precision X-form
111111 .... 10101 00100/	I	..XX			xsmaxcqp	v3.1			757	VSX Scalar Maximum Type-C Quad-Precision X-form
111111 .... 10110 00100/	I	...X	BFP128		xststdcqp	v3.0			867	VSX Scalar Test Data Class Quad-Precision X-form
111111 .... 10111 00100/	I	..XX			xsmincqp	v3.1			765	VSX Scalar Minimum Type-C Quad-Precision X-form

Table E.1: Power ISA Instruction Set Sorted by Opcode (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
111111 ..... 00000 ..... 11001 00100/	I	...X	BFP128		xsabsqp	v3.0			607	VSX Scalar Absolute Quad-Precision X-form
111111 ..... 00010 ..... 11001 00100/	I	...X	BFP128		xsxexpqp	v3.0			869	VSX Scalar Extract Exponent Quad-Precision X-form
111111 ..... 01000 ..... 11001 00100/	I	...X	BFP128		xsnabsqp	v3.0			609	VSX Scalar Negative Absolute Quad-Precision X-form
111111 ..... 10000 ..... 11001 00100/	I	...X	BFP128		xsnegqp	v3.0			610	VSX Scalar Negate Quad-Precision X-form
111111 ..... 10010 ..... 11001 00100/	I	...X	BFP128		xsxsigqp	v3.0			870	VSX Scalar Extract Significand Quad-Precision X-form
111111 ..... 11011 ..... 11001 00100.	I	...X	BFP128		xssqrtqp[o]	v3.0			638	VSX Scalar Square Root Quad-Precision [using round to Odd] X-form
111111 ..... 00000 ..... 11010 00100/	I	..XX			xscvqpuqz	v3.1			832	VSX Scalar Convert with round to zero Quad-Precision to Unsigned Quadword X-form
111111 ..... 00001 ..... 11010 00100/	I	...X	BFP128		xscvqpuwz	v3.0			834	VSX Scalar Convert with round to zero Quad-Precision to Unsigned Word format X-form
111111 ..... 00010 ..... 11010 00100/	I	...X	BFP128		xscvudqp	v3.0			852	VSX Scalar Convert Unsigned Doubleword to Quad-Precision format X-form
111111 ..... 00011 ..... 11010 00100/	I	..XX			xscvuqqp	v3.1			853	VSX Scalar Convert with round Unsigned Quadword to Quad-Precision format X-form
111111 ..... 01000 ..... 11010 00100/	I	..XX			xscvqpsqz	v3.1			826	VSX Scalar Convert with round to zero Quad-Precision to Signed Quadword X-form
111111 ..... 01001 ..... 11010 00100/	I	...X	BFP128		xscvqpswz	v3.0			828	VSX Scalar Convert with round to zero Quad-Precision to Signed Word format X-form
111111 ..... 01010 ..... 11010 00100/	I	...X	BFP128		xscvsdqp	v3.0			852	VSX Scalar Convert Signed Doubleword to Quad-Precision format X-form
111111 ..... 01011 ..... 11010 00100/	I	..XX			xscvsqqp	v3.1			853	VSX Scalar Convert with round Signed Quadword to Quad-Precision X-form
111111 ..... 10001 ..... 11010 00100/	I	...X	BFP128		xscvqpudz	v3.0			830	VSX Scalar Convert with round to zero Quad-Precision to Unsigned Doubleword format X-form
111111 ..... 10100 ..... 11010 00100.	I	...X	BFP128		xscvqpdp[o]	v3.0			791	VSX Scalar Convert with round Quad-Precision to Double-Precision format [using round to Odd] X-form
111111 ..... 10110 ..... 11010 00100/	I	...X	BFP128		xscvdpqp	v3.0			795	VSX Scalar Convert Double-Precision to Quad-Precision format X-form
111111 ..... 11001 ..... 11010 00100/	I	...X	BFP128		xscvqpsdz	v3.0			824	VSX Scalar Convert with round to zero Quad-Precision to Signed Doubleword format X-form
111111 .... 11011 00100/	I	...X	BFP128		xsiexpqp	v3.0			865	VSX Scalar Insert Exponent Quad-Precision X-form
111111 ..... ////. ....000 00101.	I	...X	BFP128		xsrqpi[x]	v3.0			806	VSX Scalar Round to Quad-Precision Integer [with Inexact] Z23-form
111111 ..... ////. 001 00101/	I	...X	BFP128		xsrqpxp	v3.0			785	VSX Scalar Round Quad-Precision to Double-Extended-Precision Z23-form
111111 ..... ///// ///// 00001 00110.	I	.XXX			mtfsb1[.]	P1			185	Move To FPSCR Bit 1 X-form
111111 ..... ///// ///// 00010 00110.	I	.XXX			mtfsb0[.]	P1			185	Move To FPSCR Bit 0 X-form
111111 ...// ////. ..../ 00100 00110.	I	.XXX			mtfsfi[.]	P1			184	Move To FPSCR Field Immediate X-form
111111 .... 11010 00110/	I	.XXX			fmrgow	v2.07			161	Floating Merge Odd Word X-form
111111 .... 11110 00110/	I	.XXX			fmrgew	v2.07			161	Floating Merge Even Word X-form
111111 ..... 00000 ///// 10010 00111.	I	.XXX			mffs[.]	P1			182	Move From FPSCR X-form
111111 ..... 00001 ///// 10010 00111/	I	.XXX			mffsce	v3.0B			182	Move From FPSCR & Clear Enables X-form
111111 ..... 10100 ..... 10010 00111/	I	.XXX			mffscdrn	v3.0B			183	Move From FPSCR Control & Set DRN X-form
111111 ..... 10101 //... 10010 00111/	I	.XXX			mffscdrni	v3.0B			183	Move From FPSCR Control & Set DRN Immediate X-form
111111 ..... 10110 ..... 10010 00111/	I	.XXX			mffscrn	v3.0B			183	Move From FPSCR Control & Set RN X-form
111111 ..... 10111 ///.. 10010 00111/	I	.XXX			mffscrni	v3.0B			183	Move From FPSCR Control & Set RN Immediate X-form
111111 ..... 11000 ///// 10010 00111/	I	.XXX			mffsl	v3.0B			184	Move From FPSCR Lightweight X-form
111111 .... 10110 00111.	I	.XXX			mtfsf[.]	P1			185	Move To FPSCR Fields XFL-form

Table E.1: Power ISA Instruction Set Sorted by Opcode (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
111111 .... 00000 01000.	I	.XXX			fcpsgn[.]	v2.05			160	Floating Copy Sign X-form
111111 ..... ///// ..... 00001 01000.	I	.XXX			fneg[.]	P1			160	Floating Negate X-form
111111 ..... ///// ..... 00010 01000.	I	.XXX			fmr[.]	P1			160	Floating Move Register X-form
111111 ..... ///// ..... 00100 01000.	I	.XXX			fnabs[.]	P1			160	Floating Negative Absolute Value X-form
111111 ..... ///// ..... 01000 01000.	I	.XXX			fabs[.]	P1			160	Floating Absolute Value X-form
111111 ..... ///// ..... 01100 01000.	I	.XXX			frin[.]	v2.02			177	Floating Round to Integer Nearest X-form
111111 ..... ///// ..... 01101 01000.	I	.XXX			friz[.]	v2.02			177	Floating Round to Integer Toward Zero X-form
111111 ..... ///// ..... 01110 01000.	I	.XXX			frip[.]	v2.02			178	Floating Round to Integer Plus X-form
111111 ..... ///// ..... 01111 01000.	I	.XXX			frim[.]	v2.02			178	Floating Round to Integer Minus X-form
111111 ..... ///// ..... 00000 01100.	I	.XXX			frsp[.]	P1			170	Floating Round to Single-Precision X-form
111111 ..... ///// ..... 00000 01110.	I	.XXX			fctiw[.]	P2			173	Floating Convert with round Double-Precision To Signed Word format X-form
111111 ..... ///// ..... 00100 01110.	I	.XXX			fctiwu[.]	v2.06			174	Floating Convert with round Double-Precision To Unsigned Word format X-form
111111 ..... ///// ..... 11001 01110.	I	.XXX			fctid[.]	PPC			171	Floating Convert with round Double-Precision To Signed Doubleword format X-form
111111 ..... ///// ..... 11010 01110.	I	.XXX			fcfid[.]	PPC			175	Floating Convert with round Signed Doubleword to Double-Precision format X-form
111111 ..... ///// ..... 11101 01110.	I	.XXX			fctidu[.]	v2.06			172	Floating Convert with round Double-Precision To Unsigned Doubleword format X-form
111111 ..... ///// ..... 11110 01110.	I	.XXX			fcfidu[.]	v2.06			175	Floating Convert with round Unsigned Doubleword to Double-Precision format X-form
111111 ..... ///// ..... 00000 01111.	I	.XXX			fctiwz[.]	P2			173	Floating Convert with truncate Double-Precision To Signed Word format X-form
111111 ..... ///// ..... 00100 01111.	I	.XXX			fctiwuz[.]	v2.06			174	Floating Convert with truncate Double-Precision To Unsigned Word format X-form
111111 ..... ///// ..... 11001 01111.	I	.XXX			fctidz[.]	PPC			171	Floating Convert with truncate Double-Precision To Signed Doubleword format X-form
111111 ..... ///// ..... 11101 01111.	I	.XXX			fctiduz[.]	v2.06			172	Floating Convert with truncate Double-Precision To Unsigned Doubleword format X-form
111111 .... ///// 10010.	I	.XXX			fdiv[.]	P1			163	Floating Divide A-form
111111 .... ///// 10100.	I	.XXX			fsub[.]	P1			162	Floating Subtract A-form
111111 .... ///// 10101.	I	.XXX			fadd[.]	P1			162	Floating Add A-form
111111 ..... ///// ..... ///// 10110.	I	.XXX			fsqrt[.]	P2			164	Floating Square Root A-form
111111 .... 10111.	I	.XXX			fsel[.]	PPC			180	Floating Select A-form
111111 ..... ///// ..... ///// 11000.	I	.XXX			fre[.]	v2.02			165	Floating Reciprocal Estimate A-form
111111 .... ///// ..... 11001.	I	.XXX			fmul[.]	P1			163	Floating Multiply A-form
111111 ..... ///// ..... ///// 11010.	I	.XXX			frsqrte[.]	PPC			166	Floating Reciprocal Square Root Estimate A-form
111111 .... 11100.	I	.XXX			fmsub[.]	P1			168	Floating Multiply-Subtract A-form
111111 .... 11101.	I	.XXX			fmadd[.]	P1			168	Floating Multiply-Add A-form
111111 .... 11110.	I	.XXX			fnmsub[.]	P1			169	Floating Negative Multiply-Subtract A-form
111111 .... 11111.	I	.XXX			fnmadd[.]	P1			169	Floating Negative Multiply-Add A-form
000001 11000 0//00 00000 00000 000000 ?????? ????? ????? ????? ????? ??????	I	..XX			pnop	v3.1			132	Prefixed Nop MRR:*-form

Appendix F. Power ISA Instruction Set Sorted by Version

This appendix lists all the instructions in the Power ISA, sorted in reverse order by ISA version, then by mnemonic. For an explanation of the values in the columns, see Appendix E.

Table F.1: Power ISA Instruction Set Sorted by Version

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
011111 .... 10111 10010.	I	....			hashchk	v3.1B			120	Hash Check X-form
011111 .... 10101 10010.	III	....			hashchkp	v3.1B	P		1099	Hash Check Privileged X-form
011111 .... 10110 10010.	I	....			hashst	v3.1B			120	Hash Store X-form
011111 .... 10100 10010.	III	....			hashstp	v3.1B	P		1099	Hash Store Privileged X-form
011111 .... ///// 00101 11011/	I	..XX			brd	v3.1			118	Byte-Reverse Doubleword X-form
011111 .... ///// 00110 11011/	I	XXXX			brh	v3.1			118	Byte-Reverse Halfword X-form
011111 .... ///// 00100 11011/	I	XXXX			brw	v3.1			118	Byte-Reverse Word X-form
011111 .... 00110 11100/	I	..XX			cfuged	v3.1			105	Centrifuge Doubleword X-form
011111 .... 00001 11011/	I	..XX			cntlzdm	v3.1			104	Count Leading Zeros Doubleword under bit Mask X-form
011111 .... 10001 11011/	I	..XX			cnttzdm	v3.1			104	Count Trailing Zeros Doubleword under bit Mask X-form
111111 ..... 00000 ..... 11111 00010/	I	...X	DFP		dcffixqq	v3.1			227	DFP Convert From Fixed Quadword Quad X-form
111111 ..... 00001 ..... 11111 00010/	I	...X	DFP		dctfixqq	v3.1			228	DFP Convert To Fixed Quadword Quad X-form
111100 ..... 11111 ..... 01011 01000.	I	..XX			lxvkq	v3.1			935	Load VSX Vector Special Value Quadword X-form
000110 0000	I	..XX			lxvp	v3.1			603	Load VSX Vector Paired DQ-form
011111 .... 01010 01101/	I	..XX			lxvpx	v3.1			604	Load VSX Vector Paired Indexed X-form
011111 .... 00000 01101.	I	..XX			lxvrbx	v3.1			584	Load VSX Vector Rightmost Byte Indexed X-form
011111 .... 00011 01101.	I	..XX			lxvrdx	v3.1			585	Load VSX Vector Rightmost Doubleword Indexed X-form
011111 .... 00001 01101.	I	..XX			lxvrhx	v3.1			586	Load VSX Vector Rightmost Halfword Indexed X-form
011111 .... 00010 01101.	I	..XX			lxvrwx	v3.1			587	Load VSX Vector Rightmost Word Indexed X-form
000100 ..... 10000 ..... 11001 000010	I	..XX			mtvsrbm	v3.1			451	Move to VSR Byte Mask VX-form
000100 .... 01010.	I	..XX			mtvsrbmi	v3.1			453	Move To VSR Byte Mask Immediate DX-form
000100 ..... 10011 ..... 11001 000010	I	..XX			mtvsrdm	v3.1			452	Move to VSR Doubleword Mask VX-form
000100 ..... 10001 ..... 11001 000010	I	..XX			mtvsrhm	v3.1			451	Move to VSR Halfword Mask VX-form
000100 ..... 10100 ..... 11001 000010	I	..XX			mtvsrqm	v3.1			453	Move to VSR Quadword Mask VX-form
000100 ..... 10010 ..... 11001 000010	I	..XX			mtvsrwm	v3.1			452	Move to VSR Word Mask VX-form
000001 100// .//.. .... 001110	I	..XX			paddi	v3.1			73	Prefixed Add Immediate MLS:D-form
011111 .... 00100 11100/	I	..XX			pdepd	v3.1			106	Parallel Bits Deposit Doubleword X-form
011111 .... 00101 11100/	I	..XX			pextd	v3.1			106	Parallel Bits Extract Doubleword X-form
000001 100// .//.. .... 100010	I	..XX			plbz	v3.1			48	Prefixed Load Byte and Zero MLS:D-form
000001 000// .//.. .... 111001	I	..XX			pld	v3.1			55	Prefixed Load Doubleword 8LS:D-form

Table F.1: Power ISA Instruction Set Sorted by Version (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
000001 100// .//.. .... 110010	I	..XX			plfd	v3.1			152	Prefixed Load Floating-Point Double MLS:D-form
000001 100// .//.. .... 110000	I	..XX			plfs	v3.1			150	Prefixed Load Floating-Point Single MLS:D-form
000001 100// .//.. .... 101010	I	..XX			plha	v3.1			51	Prefixed Load Halfword Algebraic MLS:D-form
000001 100// .//.. .... 101000	I	..XX			plhz	v3.1			50	Prefixed Load Halfword and Zero MLS:D-form
000001 000// .//.. .... 111000	I	..XX			plq	v3.1			63	Prefixed Load Quadword 8LS:D-form
000001 000// .//.. .... 101001	I	..XX			plwa	v3.1			54	Prefixed Load Word Algebraic 8LS:D-form
000001 100// .//.. .... 100000	I	..XX			plwz	v3.1			53	Prefixed Load Word and Zero MLS:D-form
000001 000// .//.. .... 101010	I	..XX			plxsd	v3.1			562	Prefixed Load VSX Scalar Doubleword 8LS:D-form
000001 000// .//.. .... 101011	I	..XX			plxssp	v3.1			567	Prefixed Load VSX Scalar Single-Precision 8LS:D-form
000001 000// .//.. .... 11001.	I	..XX			plxv	v3.1			575	Prefixed Load VSX Vector 8LS:D-form
000001 000// .//.. .... 111010	I	..XX			plxvp	v3.1			603	Prefixed Load VSX Vector Paired 8LS:D-form
000001 11100 1//// ..//// ///.. ..... 111011 .../// .... 00110 011../	I	....	MMA	MMA	pmxvbf16ger2	v3.1			889	Prefixed Masked VSX Vector bfloat16 GER (rank-2 update) MMIRR:XX3-form
000001 11100 1//// ..//// ///.. ..... 111011 .../// .... 11110 010../	I	....	MMA	MMA	pmxvbf16ger2nn	v3.1			890	Prefixed Masked VSX Vector bfloat16 GER (rank-2 update) Negative multiply, Negative accumulate MMIRR:XX3-form
000001 11100 1//// ..//// ///.. ..... 111011 .../// .... 01110 010../	I	....	MMA	MMA	pmxvbf16ger2np	v3.1			890	Prefixed Masked VSX Vector bfloat16 GER (rank-2 update) Negative multiply, Positive accumulate MMIRR:XX3-form
000001 11100 1//// ..//// ///.. ..... 111011 .../// .... 10110 010../	I	....	MMA	MMA	pmxvbf16ger2pn	v3.1			889	Prefixed Masked VSX Vector bfloat16 GER (rank-2 update) Positive multiply, Negative accumulate MMIRR:XX3-form
000001 11100 1//// ..//// ///.. ..... 111011 .../// .... 00110 010../	I	....	MMA	MMA	pmxvbf16ger2pp	v3.1			889	Prefixed Masked VSX Vector bfloat16 GER (rank-2 update) Positive multiply, Positive accumulate MMIRR:XX3-form
000001 11100 1//// ..//// ///.. ..... 111011 .../// .... 00010 011../	I	....	MMA	MMA	pmxvf16ger2	v3.1			894	Prefixed Masked VSX Vector 16-bit Floating-Point GER (rank-2 update) MMIRR:XX3-form
000001 11100 1//// ..//// ///.. ..... 111011 .../// .... 11010 010../	I	....	MMA	MMA	pmxvf16ger2nn	v3.1			895	Prefixed Masked VSX Vector 16-bit Floating-Point GER (rank-2 update) Negative multiply, Negative accumulate MMIRR:XX3-form
000001 11100 1//// ..//// ///.. ..... 111011 .../// .... 01010 010../	I	....	MMA	MMA	pmxvf16ger2np	v3.1			895	Prefixed Masked VSX Vector 16-bit Floating-Point GER (rank-2 update) Negative multiply, Positive accumulate MMIRR:XX3-form
000001 11100 1//// ..//// ///.. ..... 111011 .../// .... 10010 010../	I	....	MMA	MMA	pmxvf16ger2pn	v3.1			894	Prefixed Masked VSX Vector 16-bit Floating-Point GER (rank-2 update) Positive multiply, Negative accumulate MMIRR:XX3-form

Table F.1: Power ISA Instruction Set Sorted by Version (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
000001 11100 1//// ..//// ///.. ..... 111011 ...// .... 00010 010../	I	....	MMA	MMA	pmxvf16ger2pp	v3.1			894	Prefixed Masked VSX Vector 16-bit Floating-Point GER (rank-2 update) Positive multiply, Positive accumulate MMIRR:XX3-form
000001 11100 1//// ///// ///.. ..... 111011 ...// .... 00011 011../	I	....	MMA	MMA	pmxvf32ger	v3.1			899	Prefixed Masked VSX Vector 32-bit Floating-Point GER (rank-1 update) MMIRR:XX3-form
000001 11100 1//// ///// ///.. ..... 111011 ...// .... 11011 010../	I	....	MMA	MMA	pmxvf32gernn	v3.1			900	Prefixed Masked VSX Vector 32-bit Floating-Point GER (rank-1 update) Negative multiply, Negative accumulate MMIRR:XX3-form
000001 11100 1//// ///// ///.. ..... 111011 ...// .... 01011 010../	I	....	MMA	MMA	pmxvf32gernp	v3.1			900	Prefixed Masked VSX Vector 32-bit Floating-Point GER (rank-1 update) Negative multiply, Positive accumulate MMIRR:XX3-form
000001 11100 1//// ///// ///.. ..... 111011 ...// .... 10011 010../	I	....	MMA	MMA	pmxvf32gerpn	v3.1			899	Prefixed Masked VSX Vector 32-bit Floating-Point GER (rank-1 update) Positive multiply, Negative accumulate MMIRR:XX3-form
000001 11100 1//// ///// ///.. ..... 111011 ...// .... 00011 010../	I	....	MMA	MMA	pmxvf32gerpp	v3.1			899	Prefixed Masked VSX Vector 32-bit Floating-Point GER (rank-1 update) Positive multiply, Positive accumulate MMIRR:XX3-form
000001 11100 1//// ///// ///.. ..... 111011 ...// .... 00111 011../	I	....	MMA	MMA	pmxvf64ger	v3.1			903	Prefixed Masked VSX Vector 64-bit Floating-Point GER (rank-1 update) MMIRR:XX3-form
000001 11100 1//// ///// ///.. ..... 111011 ...// .... 11111 010../	I	....	MMA	MMA	pmxvf64gernn	v3.1			904	Prefixed Masked VSX Vector 64-bit Floating-Point GER (rank-1 update) Negative multiply, Negative accumulate MMIRR:XX3-form
000001 11100 1//// ///// ///.. ..... 111011 ...// .... 01111 010../	I	....	MMA	MMA	pmxvf64gernp	v3.1			904	Prefixed Masked VSX Vector 64-bit Floating-Point GER (rank-1 update) Negative multiply, Positive accumulate MMIRR:XX3-form
000001 11100 1//// ///// ///.. ..... 111011 ...// .... 10111 010../	I	....	MMA	MMA	pmxvf64gerpn	v3.1			903	Prefixed Masked VSX Vector 64-bit Floating-Point GER (rank-1 update) Positive multiply, Negative accumulate MMIRR:XX3-form
000001 11100 1//// ///// ///.. ..... 111011 ...// .... 00111 010../	I	....	MMA	MMA	pmxvf64gerpp	v3.1			903	Prefixed Masked VSX Vector 64-bit Floating-Point GER (rank-1 update) Positive multiply, Positive accumulate MMIRR:XX3-form
000001 11100 1//// ..//// ///.. ..... 111011 ...// .... 01001 011../	I	....	MMA	MMA	pmxvi16ger2	v3.1			878	Prefixed Masked VSX Vector 16-bit Signed Integer GER (rank-2 update) MMIRR:XX3-form
000001 11100 1//// ..//// ///.. ..... 111011 ...// .... 01101 011../	I	....	MMA	MMA	pmxvi16ger2pp	v3.1			878	Prefixed Masked VSX Vector 16-bit Signed Integer GER (rank-2 update) Positive multiply, Positive accumulate MMIRR:XX3-form
000001 11100 1//// ..//// ///.. ..... 111011 ...// .... 00101 011../	I	....	MMA	MMA	pmxvi16ger2s	v3.1			880	Prefixed Masked VSX Vector 16-bit Signed Integer GER (rank-2 update) with Saturation MMIRR:XX3-form
000001 11100 1//// ..//// ///.. ..... 111011 ...// .... 00101 010../	I	....	MMA	MMA	pmxvi16ger2spp	v3.1			880	Prefixed Masked VSX Vector 16-bit Signed Integer GER (rank-2 update) with Saturation Positive multiply, Positive accumulate MMIRR:XX3-form
000001 11100 1//// .... 111011 ...// .... 00100 011../	I	....	MMA	MMA	pmxvi4ger8	v3.1			882	Prefixed Masked VSX Vector 4-bit Signed Integer GER (rank-8 update) MMIRR:XX3-form
000001 11100 1//// .... 111011 ...// .... 00100 010../	I	....	MMA	MMA	pmxvi4ger8pp	v3.1			882	Prefixed Masked VSX Vector 4-bit Signed Integer GER (rank-8 update) Positive multiply, Positive accumulate MMIRR:XX3-form
000001 11100 1//// ..../ ///.. ..... 111011 ...// .... 00000 011../	I	....	MMA	MMA	pmxvi8ger4	v3.1			885	Prefixed Masked VSX Vector 8-bit Signed/Unsigned Integer GER (rank-4 update) MMIRR:XX3-form

Table F.1: Power ISA Instruction Set Sorted by Version (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
000001 11100 1//// ..../ ///.. ..... 111011 .../// .... 00000 010../	I	....	MMA	MMA	pmxvi8ger4pp	v3.1			885	Prefixed Masked VSX Vector 8-bit Signed/Unsigned Integer GER (rank-4 update) Positive multiply, Positive accumulate MMIRR:XX3-form
000001 11100 1//// ..../ ///.. ..... 111011 .../// .... 01100 011../	I	....	MMA	MMA	pmxvi8ger4spp	v3.1			887	Prefixed Masked VSX Vector 8-bit Signed/Unsigned Integer GER (rank-4 update) with Saturation Positive multiply, Positive accumulate MMIRR:XX3-form
000001 11000 0//00 00000 00000 000000 ?????? ????? ????? ????? ????? ??????	I	..XX			pnop	v3.1			132	Prefixed Nop MRR:*-form
000001 100// .//.. .... 100110	I	..XX			pstb	v3.1			57	Prefixed Store Byte MLS:D-form
000001 000// .//.. .... 111101	I	..XX			pstd	v3.1			60	Prefixed Store Doubleword 8LS:D-form
000001 100// .//.. .... 110110	I	..XX			pstfd	v3.1			156	Prefixed Store Floating-Point Double MLS:D-form
000001 100// .//.. .... 110100	I	..XX			pstfs	v3.1			154	Prefixed Store Floating-Point Single MLS:D-form
000001 100// .//.. .... 101100	I	..XX			psth	v3.1			58	Prefixed Store Halfword MLS:D-form
000001 000// .//.. .... 111100	I	..XX			pstq	v3.1			64	Prefixed Store Quadword 8LS:D-form
000001 100// .//.. .... 100100	I	..XX			pstw	v3.1			59	Prefixed Store Word MLS:D-form
000001 000// .//.. .... 101110	I	..XX			pstxsd	v3.1			569	Prefixed Store VSX Scalar Doubleword 8LS:D-form
000001 000// .//.. .... 101111	I	..XX			pstxssp	v3.1			573	Prefixed Store VSX Scalar Single-Precision 8LS:D-form
000001 000// .//.. .... 11011.	I	..XX			pstxv	v3.1			591	Prefixed Store VSX Vector 8LS:D-form
000001 000// .//.. .... 111110	I	..XX			pstxvp	v3.1			605	Prefixed Store VSX Vector Paired 8LS:D-form
011111 .... ///// 01100 00000/	I	XXXX			setbc	v3.1			131	Set Boolean Condition X-form
011111 .... ///// 01101 00000/	I	XXXX			setbcr	v3.1			131	Set Boolean Condition Reverse X-form
011111 .... ///// 01110 00000/	I	XXXX			setnbc	v3.1			131	Set Negative Boolean Condition X-form
011111 .... ///// 01111 00000/	I	XXXX			setnbcr	v3.1			131	Set Negative Boolean Condition Reverse X-form
000110 0001	I	..XX			stxvp	v3.1			605	Store VSX Vector Paired DQ-form
011111 .... 01110 01101/	I	..XX			stxvpx	v3.1			606	Store VSX Vector Paired Indexed X-form
011111 .... 00100 01101.	I	..XX			stxvrbx	v3.1			598	Store VSX Vector Rightmost Byte Indexed X-form
011111 .... 00111 01101.	I	..XX			stxvrdx	v3.1			598	Store VSX Vector Rightmost Doubleword Indexed X-form
011111 .... 00101 01101.	I	..XX			stxvrhx	v3.1			599	Store VSX Vector Rightmost Halfword Indexed X-form
011111 .... 00110 01101.	I	..XX			stxvrwx	v3.1			599	Store VSX Vector Rightmost Word Indexed X-form
000100 .... 10101 001101	I	..XX			vcfuged	v3.1			444	Vector Centrifuge Doubleword VX-form
000100 .... 00110 001101	I	..XX			vclrlb	v3.1			464	Vector Clear Leftmost Bytes VX-form
000100 .... 00111 001101	I	..XX			vclrrb	v3.1			464	Vector Clear Rightmost Bytes VX-form
000100 .... 11110 000100	I	..XX			vclzdm	v3.1			437	Vector Count Leading Zeros Doubleword under bit Mask VX-form
000100 .... .0111 000111	I	..XX			vcmpequq[.]	v3.1			382	Vector Compare Equal Quadword VC-form
000100 .... .1110 000111	I	..XX			vcmpgtsq[.]	v3.1			387	Vector Compare Greater Than Signed Quadword VC-form
000100 .... .1010 000111	I	..XX			vcmpgtuq[.]	v3.1			387	Vector Compare Greater Than Unsigned Quadword VC-form

Table F.1: Power ISA Instruction Set Sorted by Version (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
000100 ...// 00101 000001	I	..XX			vcmpsq	v3.1			391	Vector Compare Signed Quadword VX-form
000100 ...// 00100 000001	I	..XX			vcmpuq	v3.1			391	Vector Compare Unsigned Quadword VX-form
000100 ..... 1100. ..... 11001 000010	I	..XX			vcntmbb	v3.1			457	Vector Count Mask Bits Byte VX-form
000100 ..... 1111. ..... 11001 000010	I	..XX			vcntmbd	v3.1			458	Vector Count Mask Bits Doubleword VX-form
000100 ..... 1101. ..... 11001 000010	I	..XX			vcntmbh	v3.1			457	Vector Count Mask Bits Halfword VX-form
000100 ..... 1110. ..... 11001 000010	I	..XX			vcntmbw	v3.1			458	Vector Count Mask Bits Word VX-form
000100 .... 11111 000100	I	..XX			vctzdm	v3.1			440	Vector Count Trailing Zeros Doubleword under bit Mask VX-form
000100 .... 01111 001011	I	..XX			vdivesd	v3.1			351	Vector Divide Extended Signed Doubleword VX-form
000100 .... 01100 001011	I	..XX			vdivesq	v3.1			353	Vector Divide Extended Signed Quadword VX-form
000100 .... 01110 001011	I	..XX			vdivesw	v3.1			349	Vector Divide Extended Signed Word VX-form
000100 .... 01011 001011	I	..XX			vdiveud	v3.1			351	Vector Divide Extended Unsigned Doubleword VX-form
000100 .... 01000 001011	I	..XX			vdiveuq	v3.1			353	Vector Divide Extended Unsigned Quadword VX-form
000100 .... 01010 001011	I	..XX			vdiveuw	v3.1			349	Vector Divide Extended Unsigned Word VX-form
000100 .... 00111 001011	I	..XX			vdivsd	v3.1			350	Vector Divide Signed Doubleword VX-form
000100 .... 00100 001011	I	..XX			vdivsq	v3.1			352	Vector Divide Signed Quadword VX-form
000100 .... 00110 001011	I	..XX			vdivsw	v3.1			348	Vector Divide Signed Word VX-form
000100 .... 00011 001011	I	..XX			vdivud	v3.1			350	Vector Divide Unsigned Doubleword VX-form
000100 .... 00000 001011	I	..XX			vdivuq	v3.1			352	Vector Divide Unsigned Quadword VX-form
000100 .... 00010 001011	I	..XX			vdivuw	v3.1			348	Vector Divide Unsigned Word VX-form
000100 ..... 00000 ..... 11001 000010	I	..XX			vexpandbm	v3.1			454	Vector Expand Byte Mask VX-form
000100 ..... 00011 ..... 11001 000010	I	..XX			vexpanddm	v3.1			455	Vector Expand Doubleword Mask VX-form
000100 ..... 00001 ..... 11001 000010	I	..XX			vexpandhm	v3.1			454	Vector Expand Halfword Mask VX-form
000100 ..... 00100 ..... 11001 000010	I	..XX			vexpandqm	v3.1			456	Vector Expand Quadword Mask VX-form
000100 ..... 00010 ..... 11001 000010	I	..XX			vexpandwm	v3.1			455	Vector Expand Word Mask VX-form
000100 011110	I	..XX			vextddvlx	v3.1			302	Vector Extract Double Doubleword to VSR using GPR-specified Left-Index VA-form
000100 011111	I	..XX			vextddvrx	v3.1			302	Vector Extract Double Doubleword to VSR using GPR-specified Right-Index VA-form
000100 011000	I	..XX			vextdubvlx	v3.1			299	Vector Extract Double Unsigned Byte to VSR using GPR-specified Left-Index VA-form
000100 011001	I	..XX			vextdubvrx	v3.1			299	Vector Extract Double Unsigned Byte to VSR using GPR-specified Right-Index VA-form
000100 011010	I	..XX			vextduhvlx	v3.1			300	Vector Extract Double Unsigned Halfword to VSR using GPR-specified Left-Index VA-form
000100 011011	I	..XX			vextduhvrx	v3.1			300	Vector Extract Double Unsigned Halfword to VSR using GPR-specified Right-Index VA-form
000100 011100	I	..XX			vextduwvlx	v3.1			301	Vector Extract Double Unsigned Word to VSR using GPR-specified Left-Index VA-form
000100 011101	I	..XX			vextduwvrx	v3.1			301	Vector Extract Double Unsigned Word to VSR using GPR-specified Right-Index VA-form
000100 ..... 01000 ..... 11001 000010	I	..XX			vextractbm	v3.1			459	Vector Extract Byte Mask VX-form
000100 ..... 01011 ..... 11001 000010	I	..XX			vextractdm	v3.1			460	Vector Extract Doubleword Mask VX-form
000100 ..... 01001 ..... 11001 000010	I	..XX			vextracthm	v3.1			459	Vector Extract Halfword Mask VX-form
000100 ..... 01100 ..... 11001 000010	I	..XX			vextractqm	v3.1			461	Vector Extract Quadword Mask VX-form
000100 ..... 01010 ..... 11001 000010	I	..XX			vextractwm	v3.1			460	Vector Extract Word Mask VX-form
000100 ..... 11011 ..... 11000 000010	I	..XX			vextsd2q	v3.1			364	Vector Extend Sign Doubleword to Quadword VX-form
000100 ..... //... ..... 10011 001100	I	..XX			vgnb	v3.1			434	Vector Gather every Nth Bit VX-form

Table F.1: Power ISA Instruction Set Sorted by Version (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
000100 .... 01000 001111	I	..XX			vinsblx	v3.1			305	Vector Insert Byte from GPR using GPR-specified Left-Index VX-form
000100 .... 01100 001111	I	..XX			vinsbrx	v3.1			305	Vector Insert Byte from GPR using GPR-specified Right-Index VX-form
000100 .... 00000 001111	I	..XX			vinsbvlx	v3.1			310	Vector Insert Byte from VSR using GPR-specified Left-Index VX-form
000100 .... 00100 001111	I	..XX			vinsbvrx	v3.1			310	Vector Insert Byte from VSR using GPR-specified Right-Index VX-form
000100 ..... /.... ..... 00111 001111	I	..XX			vinsd	v3.1			309	Vector Insert Doubleword from GPR using immediate-specified index VX-form
000100 .... 01011 001111	I	..XX			vinsdlx	v3.1			308	Vector Insert Doubleword from GPR using GPR-specified Left-Index VX-form
000100 .... 01111 001111	I	..XX			vinsdrx	v3.1			308	Vector Insert Doubleword from GPR using GPR-specified Right-Index VX-form
000100 .... 01001 001111	I	..XX			vinshlx	v3.1			306	Vector Insert Halfword from GPR using GPR-specified Left-Index VX-form
000100 .... 01101 001111	I	..XX			vinshrx	v3.1			306	Vector Insert Halfword from GPR using GPR-specified Right-Index VX-form
000100 .... 00001 001111	I	..XX			vinshvlx	v3.1			311	Vector Insert Halfword from VSR using GPR-specified Left-Index VX-form
000100 .... 00101 001111	I	..XX			vinshvrx	v3.1			311	Vector Insert Halfword from VSR using GPR-specified Right-Index VX-form
000100 ..... /.... ..... 00011 001111	I	..XX			vinsw	v3.1			309	Vector Insert Word from GPR using immediate-specified index VX-form
000100 .... 01010 001111	I	..XX			vinswlx	v3.1			307	Vector Insert Word from GPR using GPR-specified Left-Index VX-form
000100 .... 01110 001111	I	..XX			vinswrx	v3.1			307	Vector Insert Word from GPR using GPR-specified Right-Index VX-form
000100 .... 00010 001111	I	..XX			vinswvlx	v3.1			312	Vector Insert Word from VSR using GPR-specified Left-Index VX-form
000100 .... 00110 001111	I	..XX			vinswvrx	v3.1			312	Vector Insert Word from VSR using GPR-specified Right-Index VX-form
000100 .... 11111 001011	I	..XX			vmodsd	v3.1			355	Vector Modulo Signed Doubleword VX-form
000100 .... 11100 001011	I	..XX			vmodsq	v3.1			356	Vector Modulo Signed Quadword VX-form
000100 .... 11110 001011	I	..XX			vmodsw	v3.1			354	Vector Modulo Signed Word VX-form
000100 .... 11011 001011	I	..XX			vmodud	v3.1			355	Vector Modulo Unsigned Doubleword VX-form
000100 .... 11000 001011	I	..XX			vmoduq	v3.1			356	Vector Modulo Unsigned Quadword VX-form
000100 .... 11010 001011	I	..XX			vmoduw	v3.1			354	Vector Modulo Unsigned Word VX-form
000100 010111	I	..XX			vmsumcud	v3.1			347	Vector Multiply-Sum & write Carry-out Unsigned Doubleword VA-form
000100 .... 01111 001000	I	..XX			vmulesd	v3.1			336	Vector Multiply Even Signed Doubleword VX-form
000100 .... 01011 001000	I	..XX			vmuleud	v3.1			335	Vector Multiply Even Unsigned Doubleword VX-form
000100 .... 01111 001001	I	..XX			vmulhsd	v3.1			339	Vector Multiply High Signed Doubleword VX-form
000100 .... 01110 001001	I	..XX			vmulhsw	v3.1			338	Vector Multiply High Signed Word VX-form
000100 .... 01011 001001	I	..XX			vmulhud	v3.1			339	Vector Multiply High Unsigned Doubleword VX-form
000100 .... 01010 001001	I	..XX			vmulhuw	v3.1			338	Vector Multiply High Unsigned Word VX-form
000100 .... 00111 001001	I	..XX			vmulld	v3.1			340	Vector Multiply Low Doubleword VX-form
000100 .... 00111 001000	I	..XX			vmulosd	v3.1			336	Vector Multiply Odd Signed Doubleword VX-form
000100 .... 00011 001000	I	..XX			vmuloud	v3.1			335	Vector Multiply Odd Unsigned Doubleword VX-form

Table F.1: Power ISA Instruction Set Sorted by Version (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
000100 .... 10111 001101	I	..XX			vpdepd	v3.1			442	Vector Parallel Bits Deposit Doubleword VX-form
000100 .... 10110 001101	I	..XX			vpextd	v3.1			443	Vector Parallel Bits Extract Doubleword VX-form
000100 .... 00000 000101	I	..XX			vrlq	v3.1			397	Vector Rotate Left Quadword VX-form
000100 .... 00001 000101	I	..XX			vrlqmi	v3.1			401	Vector Rotate Left Quadword then Mask Insert VX-form
000100 .... 00101 000101	I	..XX			vrlqnm	v3.1			399	Vector Rotate Left Quadword then AND with Mask VX-form
000100 .... 00... 010110	I	..XX			vsldbi	v3.1			289	Vector Shift Left Double by Bit Immediate VN-form
000100 .... 00100 000101	I	..XX			vslq	v3.1			404	Vector Shift Left Quadword VX-form
000100 .... 01100 000101	I	..XX			vsraq	v3.1			410	Vector Shift Right Algebraic Quadword VX-form
000100 .... 01... 010110	I	..XX			vsrdbi	v3.1			289	Vector Shift Right Double by Bit Immediate VN-form
000100 .... 01000 000101	I	..XX			vsrq	v3.1			407	Vector Shift Right Quadword VX-form
000100 ..... 00000 ..... .0000 001101	I	..XX			vstribl[.]	v3.1			462	Vector String Isolate Byte Left-justified VC-form
000100 ..... 00001 ..... .0000 001101	I	..XX			vstribr[.]	v3.1			462	Vector String Isolate Byte Right-justified VC-form
000100 ..... 00010 ..... .0000 001101	I	..XX			vstrihl[.]	v3.1			463	Vector String Isolate Halfword Left-justified VC-form
000100 ..... 00011 ..... .0000 001101	I	..XX			vstrihr[.]	v3.1			463	Vector String Isolate Halfword Right-justified VC-form
111111 .... 00010 00100/	I	..XX			xscmpeqqp	v3.1			750	VSX Scalar Compare Equal Quad-Precision X-form
111111 .... 00110 00100/	I	..XX			xscmpgeqp	v3.1			752	VSX Scalar Compare Greater Than or Equal Quad-Precision X-form
111111 .... 00111 00100/	I	..XX			xscmpgtqp	v3.1			754	VSX Scalar Compare Greater Than Quad-Precision X-form
111111 ..... 01000 ..... 11010 00100/	I	..XX			xscvqpsqz	v3.1			826	VSX Scalar Convert with round to zero Quad-Precision to Signed Quadword X-form
111111 ..... 00000 ..... 11010 00100/	I	..XX			xscvqpuqz	v3.1			832	VSX Scalar Convert with round to zero Quad-Precision to Unsigned Quadword X-form
111111 ..... 01011 ..... 11010 00100/	I	..XX			xscvsqqp	v3.1			853	VSX Scalar Convert with round Signed Quadword to Quad-Precision X-form
111111 ..... 00011 ..... 11010 00100/	I	..XX			xscvuqqp	v3.1			853	VSX Scalar Convert with round Unsigned Quadword to Quad-Precision format X-form
111111 .... 10101 00100/	I	..XX			xsmaxcqp	v3.1			757	VSX Scalar Maximum Type-C Quad-Precision X-form
111111 .... 10111 00100/	I	..XX			xsmincqp	v3.1			765	VSX Scalar Minimum Type-C Quad-Precision X-form
111011 ...// .... 00110 011../	I	....	MMA	MMA	xvbf16ger2	v3.1			889	VSX Vector bfloat16 GER (rank-2 update) XX3-form
111011 ...// .... 11110 010../	I	....	MMA	MMA	xvbf16ger2nn	v3.1			889	VSX Vector bfloat16 GER (rank-2 update) Negative multiply, Negative accumulate XX3-form
111011 ...// .... 01110 010../	I	....	MMA	MMA	xvbf16ger2np	v3.1			889	VSX Vector bfloat16 GER (rank-2 update) Negative multiply, Positive accumulate XX3-form
111011 ...// .... 10110 010../	I	....	MMA	MMA	xvbf16ger2pn	v3.1			889	VSX Vector bfloat16 GER (rank-2 update) Positive multiply, Negative accumulate XX3-form
111011 ...// .... 00110 010../	I	....	MMA	MMA	xvbf16ger2pp	v3.1			889	VSX Vector bfloat16 GER (rank-2 update) Positive multiply, Positive accumulate XX3-form
111100 ..... 10000 ..... 11101 1011..	I	..XX			xvcvbf16spn	v3.1			800	VSX Vector Convert bfloat16 to Single-Precision format Non-signaling XX2-form
111100 ..... 10001 ..... 11101 1011..	I	..XX			xvcvspbf16	v3.1			792	VSX Vector Convert with round Single-Precision to bfloat16 format XX2-form
111011 ...// .... 00010 011../	I	....	MMA	MMA	xvf16ger2	v3.1			894	VSX Vector 16-bit Floating-Point GER (rank-2 update) XX3-form
111011 ...// .... 11010 010../	I	....	MMA	MMA	xvf16ger2nn	v3.1			894	VSX Vector 16-bit Floating-Point GER (rank-2 update) Negative multiply, Negative accumulate XX3-form
111011 ...// .... 01010 010../	I	....	MMA	MMA	xvf16ger2np	v3.1			894	VSX Vector 16-bit Floating-Point GER (rank-2 update) Negative multiply, Positive accumulate XX3-form

Table F.1: Power ISA Instruction Set Sorted by Version (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
111011 ...// .... 10010 010../	I	....	MMA	MMA	xvf16ger2pn	v3.1			894	VSX Vector 16-bit Floating-Point GER (rank-2 update) Positive multiply, Negative accumulate XX3-form
111011 ...// .... 00010 010../	I	....	MMA	MMA	xvf16ger2pp	v3.1			894	VSX Vector 16-bit Floating-Point GER (rank-2 update) Positive multiply, Positive accumulate XX3-form
111011 ...// .... 00011 011../	I	....	MMA	MMA	xvf32ger	v3.1			899	VSX Vector 32-bit Floating-Point GER (rank-1 update) XX3-form
111011 ...// .... 11011 010../	I	....	MMA	MMA	xvf32gernn	v3.1			899	VSX Vector 32-bit Floating-Point GER (rank-1 update) Negative multiply, Negative accumulate XX3-form
111011 ...// .... 01011 010../	I	....	MMA	MMA	xvf32gernp	v3.1			899	VSX Vector 32-bit Floating-Point GER (rank-1 update) Negative multiply, Positive accumulate XX3-form
111011 ...// .... 10011 010../	I	....	MMA	MMA	xvf32gerpn	v3.1			899	VSX Vector 32-bit Floating-Point GER (rank-1 update) Positive multiply, Negative accumulate XX3-form
111011 ...// .... 00011 010../	I	....	MMA	MMA	xvf32gerpp	v3.1			899	VSX Vector 32-bit Floating-Point GER (rank-1 update) Positive multiply, Positive accumulate XX3-form
111011 ...// .... 00111 011../	I	....	MMA	MMA	xvf64ger	v3.1			903	VSX Vector 64-bit Floating-Point GER (rank-1 update) XX3-form
111011 ...// .... 11111 010../	I	....	MMA	MMA	xvf64gernn	v3.1			903	VSX Vector 64-bit Floating-Point GER (rank-1 update) Negative multiply, Negative accumulate XX3-form
111011 ...// .... 01111 010../	I	....	MMA	MMA	xvf64gernp	v3.1			903	VSX Vector 64-bit Floating-Point GER (rank-1 update) Negative multiply, Positive accumulate XX3-form
111011 ...// .... 10111 010../	I	....	MMA	MMA	xvf64gerpn	v3.1			903	VSX Vector 64-bit Floating-Point GER (rank-1 update) Positive multiply, Negative accumulate XX3-form
111011 ...// .... 00111 010../	I	....	MMA	MMA	xvf64gerpp	v3.1			903	VSX Vector 64-bit Floating-Point GER (rank-1 update) Positive multiply, Positive accumulate XX3-form
111011 ...// .... 01001 011../	I	....	MMA	MMA	xvi16ger2	v3.1			878	VSX Vector 16-bit Signed Integer GER (rank-2 update) XX3-form
111011 ...// .... 01101 011../	I	....	MMA	MMA	xvi16ger2pp	v3.1			878	VSX Vector 16-bit Signed Integer GER (rank-2 update) Positive multiply, Positive accumulate XX3-form
111011 ...// .... 00101 011../	I	....	MMA	MMA	xvi16ger2s	v3.1			880	VSX Vector 16-bit Signed Integer GER (rank-2 update) with Saturation XX3-form
111011 ...// .... 00101 010../	I	....	MMA	MMA	xvi16ger2spp	v3.1			880	VSX Vector 16-bit Signed Integer GER (rank-2 update) with Saturation Positive multiply, Positive accumulate XX3-form
111011 ...// .... 00100 011../	I	....	MMA	MMA	xvi4ger8	v3.1			882	VSX Vector 4-bit Signed Integer GER (rank-8 update) XX3-form
111011 ...// .... 00100 010../	I	....	MMA	MMA	xvi4ger8pp	v3.1			882	VSX Vector 4-bit Signed Integer GER (rank-8 update) Positive multiply, Positive accumulate XX3-form
111011 ...// .... 00000 011../	I	....	MMA	MMA	xvi8ger4	v3.1			885	VSX Vector 8-bit Signed/Unsigned Integer GER (rank-4 update) XX3-form
111011 ...// .... 00000 010../	I	....	MMA	MMA	xvi8ger4pp	v3.1			885	VSX Vector 8-bit Signed/Unsigned Integer GER (rank-4 update) Positive multiply, Positive accumulate XX3-form
111011 ...// .... 01100 011../	I	....	MMA	MMA	xvi8ger4spp	v3.1			887	VSX Vector 8-bit Signed/Unsigned Integer GER (rank-4 update) with Saturation Positive multiply, Positive accumulate XX3-form
111100 ...// 00010 ..... 11101 1011./	I	..XX			xvtlsbb	v3.1			936	VSX Vector Test Least-Significant Bit by Byte XX2-form
000001 01000 0//// ///// ///// ////// 100001 .... 00....	I	..XX			xxblendvb	v3.1			912	VSX Vector Blend Variable Byte 8RR:XX4-form
000001 01000 0//// ///// ///// ////// 100001 .... 11....	I	..XX			xxblendvd	v3.1			912	VSX Vector Blend Variable Doubleword 8RR:XX4-form

Table F.1: Power ISA Instruction Set Sorted by Version (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
000001 01000 0//// ///// ///// ////// 100001 .... 01....	I	..XX			xxblendvh	v3.1			913	VSX Vector Blend Variable Halfword 8RR:XX4-form
000001 01000 0//// ///// ///// ////// 100001 .... 10....	I	..XX			xxblendvw	v3.1			913	VSX Vector Blend Variable Word 8RR:XX4-form
000001 01000 0//// ///// ///.. ..... 100010 .... 01....	I	..XX			xxeval	v3.1			910	VSX Vector Evaluate 8RR:XX4-form
111100 .... 11100 10100.	I	..XX			xxgenpcvbm	v3.1			926	VSX Vector Generate PCV from Byte Mask X-form
111100 .... 11101 10101.	I	..XX			xxgenpcvdm	v3.1			928	VSX Vector Generate PCV from Doubleword Mask X-form
111100 .... 11100 10101.	I	..XX			xxgenpcvhm	v3.1			931	VSX Vector Generate PCV from Halfword Mask X-form
111100 .... 11101 10100.	I	..XX			xxgenpcvwm	v3.1			933	VSX Vector Generate PCV from Word Mask X-form
011111 .../// 00000 ///// 00101 10001/	I	....	MMA	MMA	xxmfacc	v3.1			876	VSX Move From Accumulator X-form
011111 .../// 00001 ///// 00101 10001/	I	....	MMA	MMA	xxmtacc	v3.1			877	VSX Move To Accumulator X-form
000001 01000 0//// ///// ///// ///... 100010 .... 00....	I	..XX			xxpermx	v3.1			924	VSX Vector Permute Extended 8RR:XX4-form
011111 .../// 00011 ///// 00101 10001/	I	....	MMA	MMA	xxsetaccz	v3.1			877	VSX Set Accumulator to Zero X-form
000001 01000 0//// .... 100000 ..... 000..	I	..XX			xxsplti32dx	v3.1			919	VSX Vector Splat Immediate32 Doubleword Indexed 8RR:D-form
000001 01000 0//// .... 100000 ..... 0010.	I	..XX			xxspltidp	v3.1			920	VSX Vector Splat Immediate Double-Precision 8RR:D-form
000001 01000 0//// .... 100000 ..... 0011.	I	..XX			xxspltiw	v3.1			920	VSX Vector Splat Immediate Word 8RR:D-form
011111 ///// ///// 00011 01110/	III	....			msgclru	v3.0C	UV		1274	Message Clear Ultravisor X-form
011111 ///// ///// 00010 01110/	III	....			msgsndu	v3.0C	UV		1274	Message Send Ultravisor X-form
010011 ///// ///// ///// 01001 10010/	III	....			urfid	v3.0C	UV		1087	Ultravisor Return From Interrupt Doubleword XL-form
011111 ....101 01010/	I	XXXX			addex	v3.0B			78	Add Extended using alternate carry bit Z23-form
111111 ..... 10100 ..... 10010 00111/	I	.XXX			mffscdrn	v3.0B			183	Move From FPSCR Control & Set DRN X-form
111111 ..... 10101 //... 10010 00111/	I	.XXX			mffscdrni	v3.0B			183	Move From FPSCR Control & Set DRN Immediate X-form
111111 ..... 00001 ///// 10010 00111/	I	.XXX			mffsce	v3.0B			182	Move From FPSCR & Clear Enables X-form
111111 ..... 10110 ..... 10010 00111/	I	.XXX			mffscrn	v3.0B			183	Move From FPSCR Control & Set RN X-form
111111 ..... 10111 ///.. 10010 00111/	I	.XXX			mffscrni	v3.0B			183	Move From FPSCR Control & Set RN Immediate X-form
111111 ..... 11000 ///// 10010 00111/	I	.XXX			mffsl	v3.0B			184	Move From FPSCR Lightweight X-form
011111 ..... ////. ///// 11010 10010/	III	...X			slbiag	v3.0B	P		1167	SLB Invalidate All Global X-form
000100 100011	I	..XX			vmsumudm	v3.0B			346	Vector Multiply-Sum Unsigned Doubleword Modulo VA-form
010011 .... 00010.	I	XXXX			addpcis	v3.0			74	Add PC Immediate Shifted DX-form
000100 ..... 00111 ..... 1.110 000001	I	..XX			bcdcfn.	v3.0			468	Decimal Convert From National VX-form
000100 ..... 00010 ..... 1.110 000001	I	..XX			bcdcfsq.	v3.0			472	Decimal Convert From Signed Quadword VX-form
000100 ..... 00110 ..... 1.110 000001	I	..XX			bcdcfz.	v3.0			469	Decimal Convert From Zoned VX-form
000100 .... 01101 000001	I	..XX			bcdcpsgn.	v3.0			476	Decimal Copy Sign VX-form
000100 ..... 00101 ..... 1/110 000001	I	..XX			bcdctn.	v3.0			470	Decimal Convert To National VX-form
000100 ..... 00000 ..... 1/110 000001	I	..XX			bcdctsq.	v3.0			473	Decimal Convert To Signed Quadword VX-form
000100 ..... 00100 ..... 1.110 000001	I	..XX			bcdctz.	v3.0			471	Decimal Convert To Zoned VX-form
000100 .... 1.011 000001	I	..XX			bcds.	v3.0			478	Decimal Shift VX-form
000100 ..... 11111 ..... 1.110 000001	I	..XX			bcdsetsgn.	v3.0			477	Decimal Set Sign VX-form
000100 .... 1.111 000001	I	..XX			bcdsr.	v3.0			480	Decimal Shift & Round VX-form
000100 .... 1.100 000001	I	..XX			bcdtrunc.	v3.0			481	Decimal Truncate VX-form
000100 .... 1/010 000001	I	..XX			bcdus.	v3.0			479	Decimal Unsigned Shift VX-form
000100 .... 1/101 000001	I	..XX			bcdutrunc.	v3.0			482	Decimal Unsigned Truncate VX-form
011111 .../// 00111 00000/	I	XXXX			cmpeqb	v3.0			93	Compare Equal Byte X-form

Table F.1: Power ISA Instruction Set Sorted by Version (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
011111 …/. 00110 00000/	I	XXXX			cmprb	v3.0			92	Compare Ranged Byte X-form
011111 .... ///// 10001 11010.	I	..XX			cnttzd[.]	v3.0			104	Count Trailing Zeros Doubleword X-form
011111 .... ///// 10000 11010.	I	XXXX			cnttzw[.]	v3.0			101	Count Trailing Zeros Word X-form
011111 ////1 .... 11000 00110/	II	....			copy	v3.0			1007	Copy X-form
011111 ///// ///// ///// 11010 00110/	II	....			cpabort	v3.0			1008	Copy-Paste Abort X-form
011111 ..... ///.. ///// 10111 10011/	I	XXXX			darn	v3.0			84	Deliver A Random Number X-form
111011 …// 10101 00011/	I	…X	DFP		dtstsfi	v3.0			213	DFP Test Significance Immediate X-form
111111 …// 10101 00011/	I	…X	DFP		dtstsfiq	v3.0			213	DFP Test Significance Immediate Quad X-form
011111 .... 11011 1101.	I	..XX			extswsli[.]	v3.0			116	Extend Sign Word and Shift Left Immediate XS-form
011111 .... 10011 00110/	II	…X	AMO		ldat	v3.0			1011	Load Doubleword Atomic X-form
011111 .... 10010 00110/	II	…X	AMO		lwat	v3.0			1011	Load Word Atomic X-form
111001 10	I	..XX			lxsd	v3.0			562	Load VSX Scalar Doubleword DS-form
011111 .... 11000 01101.	I	..XX			lxsibzx	v3.0			564	Load VSX Scalar as Integer Byte & Zero Indexed X-form
011111 .... 11001 01101.	I	..XX			lxsihzx	v3.0			564	Load VSX Scalar as Integer Halfword & Zero Indexed X-form
111001 11	I	..XX			lxssp	v3.0			567	Load VSX Scalar Single-Precision DS-form
111101 001	I	..XX			lxv	v3.0			575	Load VSX Vector DQ-form
011111 .... 11011 01100.	I	..XX			lxvb16x	v3.0			576	Load VSX Vector Byte*16 Indexed X-form
011111 .... 11001 01100.	I	..XX			lxvh8x	v3.0			578	Load VSX Vector Halfword*8 Indexed X-form
011111 .... 01000 01101.	I	..XX			lxvl	v3.0			588	Load VSX Vector with Length X-form
011111 .... 01001 01101.	I	..XX			lxvll	v3.0			590	Load VSX Vector with Length Left-justified X-form
011111 .... 01011 01100.	I	..XX			lxvwsx	v3.0			583	Load VSX Vector Word & Splat Indexed X-form
011111 .... 0100/ 01100.	I	..XX			lxvx	v3.0			580	Load VSX Vector Indexed X-form
000100 110000	I	..XX			maddhd	v3.0			86	Multiply-Add High Doubleword VA-form
000100 110001	I	..XX			maddhdu	v3.0			86	Multiply-Add High Doubleword Unsigned VA-form
000100 110011	I	..XX			maddld	v3.0			86	Multiply-Add Low Doubleword VA-form
011111 …// ///// ///// 10010 00000/	I	XXXX			mcrxrx	v3.0			129	Move to CR from XER Extended X-form
011111 .... ///// 01001 10011.	I	..XX			mfvsrld	v3.0			122	Move From VSR Lower Doubleword X-form
011111 .... 11000 01001/	I	..XX			modsd	v3.0			89	Modulo Signed Doubleword X-form
011111 .... 11000 01011/	I	XXXX			modsw	v3.0			83	Modulo Signed Word X-form
011111 .... 01000 01001/	I	..XX			modud	v3.0			89	Modulo Unsigned Doubleword X-form
011111 .... 01000 01011/	I	XXXX			moduw	v3.0			83	Modulo Unsigned Word X-form
011111 ///// ///// ///// 11011 10110/	III	..XX			msgsync	v3.0	HV		1277	Message Synchronize X-form
011111 .... 01101 10011.	I	..XX			mtvsrdd	v3.0			125	Move To VSR Double Doubleword X-form
011111 .... ///// 01100 10011.	I	..XX			mtvsrws	v3.0			125	Move To VSR Word & Splat X-form
011111 ////. .... 11100 00110.	II	....			paste.	v3.0			1008	Paste X-form
010011 ///// ///// ///// 00010 10010/	III	XXXX			rfscv	v3.0	P		1085	Return From System Call Vectored XL-form
010001 ///// ///// ////. ..... .///01	I	XXXX			scv	v3.0			45	System Call Vectored SC-form
011111 ..... …// ///// 00100 00000/	I	XXXX			setb	v3.0			131	Set Boolean X-form
011111 ..... ///// 01110 10010/	III	…X			slbieg	v3.0	P		1162	SLB Invalidate Entry Global X-form
011111 ///// ///// ///// 01010 10010/	III	…X			slbsync	v3.0	P		1171	SLB Synchronize X-form
011111 .... 10111 00110/	II	…X	AMO		stdat	v3.0			1013	Store Doubleword Atomic X-form
010011 ///// ///// ///// 01011 10010/	III	..XX			stop	v3.0	P		1089	Stop XL-form
011111 .... 10110 00110/	II	…X	AMO		stwat	v3.0			1013	Store Word Atomic X-form
111101 10	I	..XX			stxsd	v3.0			569	Store VSX Scalar Doubleword DS-form
011111 .... 11100 01101.	I	..XX			stxsibx	v3.0			571	Store VSX Scalar as Integer Byte Indexed X-form
011111 .... 11101 01101.	I	..XX			stxsihx	v3.0			571	Store VSX Scalar as Integer Halfword Indexed X-form
111101 11	I	..XX			stxssp	v3.0			573	Store VSX Scalar Single-Precision DS-form

Table F.1: Power ISA Instruction Set Sorted by Version (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
111101 101	I	..XX			stxv	v3.0			591	Store VSX Vector DQ-form
011111 .... 11111 01100.	I	..XX			stxvb16x	v3.0			592	Store VSX Vector Byte*16 Indexed X-form
011111 .... 11101 01100.	I	..XX			stxvh8x	v3.0			594	Store VSX Vector Halfword*8 Indexed X-form
011111 .... 01100 01101.	I	..XX			stxvl	v3.0			600	Store VSX Vector with Length X-form
011111 .... 01101 01101.	I	..XX			stxvll	v3.0			602	Store VSX Vector with Length Left-justified X-form
011111 .... 01100 01100.	I	..XX			stxvx	v3.0			596	Store VSX Vector Indexed X-form
000100 .... 10000 000011	I	..XX			vabsdub	v3.0			368	Vector Absolute Difference Unsigned Byte VX-form
000100 .... 10001 000011	I	..XX			vabsduh	v3.0			368	Vector Absolute Difference Unsigned Halfword VX-form
000100 .... 10010 000011	I	..XX			vabsduw	v3.0			369	Vector Absolute Difference Unsigned Word VX-form
000100 .... 10111 001100	I	..XX			vbpermd	v3.0			449	Vector Bit Permute Doubleword VX-form
000100 ..... 00000 ..... 11000 000010	I	..XX			vclzlsbb	v3.0			441	Vector Count Leading Zero Least-Significant Bits Byte VX-form
000100 .... .0000 000111	I	..XX			vcmpneb[.]	v3.0			388	Vector Compare Not Equal Byte VC-form
000100 .... .0001 000111	I	..XX			vcmpneh[.]	v3.0			389	Vector Compare Not Equal Halfword VC-form
000100 .... .0010 000111	I	..XX			vcmpnew[.]	v3.0			390	Vector Compare Not Equal Word VC-form
000100 .... .0100 000111	I	..XX			vcmpnezb[.]	v3.0			388	Vector Compare Not Equal or Zero Byte VC-form
000100 .... .0101 000111	I	..XX			vcmpnezh[.]	v3.0			389	Vector Compare Not Equal or Zero Halfword VC-form
000100 .... .0110 000111	I	..XX			vcmpnezw[.]	v3.0			390	Vector Compare Not Equal or Zero Word VC-form
000100 ..... 11100 ..... 11000 000010	I	..XX			vctzb	v3.0			438	Vector Count Trailing Zeros Byte VX-form
000100 ..... 11111 ..... 11000 000010	I	..XX			vctzd	v3.0			440	Vector Count Trailing Zeros Doubleword VX-form
000100 ..... 11101 ..... 11000 000010	I	..XX			vctzh	v3.0			438	Vector Count Trailing Zeros Halfword VX-form
000100 ..... 00001 ..... 11000 000010	I	..XX			vctzlsbb	v3.0			441	Vector Count Trailing Zero Least-Significant Bits Byte VX-form
000100 ..... 11110 ..... 11000 000010	I	..XX			vctzw	v3.0			439	Vector Count Trailing Zeros Word VX-form
000100 ..... /.... ..... 01011 001101	I	..XX			vextractd	v3.0			295	Vector Extract Doubleword to VSR using immediate-specified index VX-form
000100 ..... /.... ..... 01000 001101	I	..XX			vextractub	v3.0			294	Vector Extract Unsigned Byte to VSR using immediate-specified index VX-form
000100 ..... /.... ..... 01001 001101	I	..XX			vextractuh	v3.0			294	Vector Extract Unsigned Halfword to VSR using immediate-specified index VX-form
000100 ..... /.... ..... 01010 001101	I	..XX			vextractuw	v3.0			295	Vector Extract Unsigned Word to VSR using immediate-specified index VX-form
000100 ..... 11000 ..... 11000 000010	I	..XX			vextsb2d	v3.0			363	Vector Extend Sign Byte To Doubleword VX-form
000100 ..... 10000 ..... 11000 000010	I	..XX			vextsb2w	v3.0			362	Vector Extend Sign Byte To Word VX-form
000100 ..... 11001 ..... 11000 000010	I	..XX			vextsh2d	v3.0			363	Vector Extend Sign Halfword To Doubleword VX-form
000100 ..... 10001 ..... 11000 000010	I	..XX			vextsh2w	v3.0			362	Vector Extend Sign Halfword To Word VX-form
000100 ..... 11010 ..... 11000 000010	I	..XX			vextsw2d	v3.0			364	Vector Extend Sign Word To Doubleword VX-form
000100 .... 11000 001101	I	..XX			vextublx	v3.0			296	Vector Extract Unsigned Byte to GPR using GPR-specified Left-Index VX-form
000100 .... 11100 001101	I	..XX			vextubrx	v3.0			296	Vector Extract Unsigned Byte to GPR using GPR-specified Right-Index VX-form
000100 .... 11001 001101	I	..XX			vextuhlx	v3.0			297	Vector Extract Unsigned Halfword to GPR using GPR-specified Left-Index VX-form
000100 .... 11101 001101	I	..XX			vextuhrx	v3.0			297	Vector Extract Unsigned Halfword to GPR using GPR-specified Right-Index VX-form
000100 .... 11010 001101	I	..XX			vextuwlx	v3.0			298	Vector Extract Unsigned Word to GPR using GPR-specified Left-Index VX-form
000100 .... 11110 001101	I	..XX			vextuwrx	v3.0			298	Vector Extract Unsigned Word to GPR using GPR-specified Right-Index VX-form

Table F.1: Power ISA Instruction Set Sorted by Version (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
000100 ..... /.... ..... 01100 001101	I	..XX			vinsertb	v3.0			303	Vector Insert Byte from VSR using immediate-specified index VX-form
000100 ..... /.... ..... 01111 001101	I	..XX			vinsertd	v3.0			304	Vector Insert Doubleword from VSR using immediate-specified index VX-form
000100 ..... /.... ..... 01101 001101	I	..XX			vinserth	v3.0			303	Vector Insert Halfword from VSR using immediate-specified index VX-form
000100 ..... /.... ..... 01110 001101	I	..XX			vinsertw	v3.0			304	Vector Insert Word from VSR using immediate-specified index VX-form
000100 .... ///// 00000 000001	I	..XX			vmul10cuq	v3.0			474	Vector Multiply-by-10 & write Carry-out Unsigned Quadword VX-form
000100 .... 00001 000001	I	..XX			vmul10ecuq	v3.0			475	Vector Multiply-by-10 Extended & write Carry-out Unsigned Quadword VX-form
000100 .... 01001 000001	I	..XX			vmul10euq	v3.0			475	Vector Multiply-by-10 Extended Unsigned Quadword VX-form
000100 .... ///// 01000 000001	I	..XX			vmul10uq	v3.0			474	Vector Multiply-by-10 Unsigned Quadword VX-form
000100 ..... 00111 ..... 11000 000010	I	..XX			vnegd	v3.0			361	Vector Negate Doubleword VX-form
000100 ..... 00110 ..... 11000 000010	I	..XX			vnegw	v3.0			361	Vector Negate Word VX-form
000100 111011	I	..XX			vpermr	v3.0			286	Vector Permute Right-indexed VA-form
000100 ..... 01001 ..... 11000 000010	I	..XX			vprtybd	v3.0			447	Vector Parity Byte Doubleword VX-form
000100 ..... 01010 ..... 11000 000010	I	..XX			vprtybq	v3.0			448	Vector Parity Byte Quadword VX-form
000100 ..... 01000 ..... 11000 000010	I	..XX			vprtybw	v3.0			447	Vector Parity Byte Word VX-form
000100 .... 00011 000101	I	..XX			vrldmi	v3.0			401	Vector Rotate Left Doubleword then Mask Insert VX-form
000100 .... 00111 000101	I	..XX			vrldnm	v3.0			398	Vector Rotate Left Doubleword then AND with Mask VX-form
000100 .... 00010 000101	I	..XX			vrlwmi	v3.0			400	Vector Rotate Left Word then Mask Insert VX-form
000100 .... 00110 000101	I	..XX			vrlwnm	v3.0			398	Vector Rotate Left Word then AND with Mask VX-form
000100 .... 11101 000100	I	..XX			vslv	v3.0			292	Vector Shift Left Variable VX-form
000100 .... 11100 000100	I	..XX			vsrv	v3.0			292	Vector Shift Right Variable VX-form
111111 ..... 00000 ..... 11001 00100/	I	...X	BFP128		xsabsqp	v3.0			607	VSX Scalar Absolute Quad-Precision X-form
111111 .... 00000 00100.	I	...X	BFP128		xsaddqp[o]	v3.0			620	VSX Scalar Add Quad-Precision [using round to Odd] X-form
111100 .... 00000 011...	I	..XX			xscmpeqdp	v3.0			749	VSX Scalar Compare Equal Double-Precision XX3-form
111100 ...// .... 00111 011../	I	..XX			xscmpexpdp	v3.0			862	VSX Scalar Compare Exponents Double-Precision XX3-form
111111 ...// 00101 00100/	I	...X	BFP128		xscmpexpqp	v3.0			863	VSX Scalar Compare Exponents Quad-Precision X-form
111100 .... 00010 011...	I	..XX			xscmpgedp	v3.0			751	VSX Scalar Compare Greater Than or Equal Double-Precision XX3-form
111100 .... 00001 011...	I	..XX			xscmpgtdp	v3.0			753	VSX Scalar Compare Greater Than Double-Precision XX3-form
111111 ...// 00100 00100/	I	...X	BFP128		xscmpoqp	v3.0			745	VSX Scalar Compare Ordered Quad-Precision X-form
111111 ...// 10100 00100/	I	...X	BFP128		xscmpuqp	v3.0			748	VSX Scalar Compare Unordered Quad-Precision X-form
111111 .... 00011 00100/	I	...X	BFP128		xscpsgnqp	v3.0			608	VSX Scalar Copy Sign Quad-Precision X-form
111100 ..... 10001 ..... 10101 1011..	I	..XX			xscvdphp	v3.0			788	VSX Scalar Convert with round Double-Precision to Half-Precision format XX2-form
111111 ..... 10110 ..... 11010 00100/	I	...X	BFP128		xscvdpqp	v3.0			795	VSX Scalar Convert Double-Precision to Quad-Precision format X-form
111100 ..... 10000 ..... 10101 1011..	I	..XX			xscvhpdp	v3.0			796	VSX Scalar Convert Half-Precision to Double-Precision format XX2-form

Table F.1: Power ISA Instruction Set Sorted by Version (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
111111 ..... 10100 ..... 11010 00100.	I	...X	BFP128		xscvqpdp[o]	v3.0			791	VSX Scalar Convert with round Quad-Precision to Double-Precision format [using round to Odd] X-form
111111 ..... 11001 ..... 11010 00100/	I	...X	BFP128		xscvqpsdz	v3.0			824	VSX Scalar Convert with round to zero Quad-Precision to Signed Doubleword format X-form
111111 ..... 01001 ..... 11010 00100/	I	...X	BFP128		xscvqpswz	v3.0			828	VSX Scalar Convert with round to zero Quad-Precision to Signed Word format X-form
111111 ..... 10001 ..... 11010 00100/	I	...X	BFP128		xscvqpudz	v3.0			830	VSX Scalar Convert with round to zero Quad-Precision to Unsigned Doubleword format X-form
111111 ..... 00001 ..... 11010 00100/	I	...X	BFP128		xscvqpuwz	v3.0			834	VSX Scalar Convert with round to zero Quad-Precision to Unsigned Word format X-form
111111 ..... 01010 ..... 11010 00100/	I	...X	BFP128		xscvsdqp	v3.0			852	VSX Scalar Convert Signed Doubleword to Quad-Precision format X-form
111111 ..... 00010 ..... 11010 00100/	I	...X	BFP128		xscvudqp	v3.0			852	VSX Scalar Convert Unsigned Doubleword to Quad-Precision format X-form
111111 .... 10001 00100.	I	...X	BFP128		xsdivqp[o]	v3.0			626	VSX Scalar Divide Quad-Precision [using round to Odd] X-form
111100 .... 11100 10110.	I	..XX			xsiexpdp	v3.0			864	VSX Scalar Insert Exponent Double-Precision X-form
111111 .... 11011 00100/	I	...X	BFP128		xsiexpqp	v3.0			865	VSX Scalar Insert Exponent Quad-Precision X-form
111111 .... 01100 00100.	I	...X	BFP128		xsmaddqp[o]	v3.0			654	VSX Scalar Multiply-Add Quad-Precision [using round to Odd] X-form
111100 .... 10000 000...	I	..XX			xsmaxcdp	v3.0			755	VSX Scalar Maximum Type-C Double-Precision XX3-form
111100 .... 10010 000...	I	..XX			xsmaxjdp	v3.0			761	VSX Scalar Maximum Type-J Double-Precision XX3-form
111100 .... 10001 000...	I	..XX			xsmincdp	v3.0			763	VSX Scalar Minimum Type-C Double-Precision XX3-form
111100 .... 10011 000...	I	..XX			xsminjdp	v3.0			769	VSX Scalar Minimum Type-J Double-Precision XX3-form
111111 .... 01101 00100.	I	...X	BFP128		xsmsubqp[o]	v3.0			663	VSX Scalar Multiply-Subtract Quad-Precision [using round to Odd] X-form
111111 .... 00001 00100.	I	...X	BFP128		xsmulqp[o]	v3.0			632	VSX Scalar Multiply Quad-Precision [using round to Odd] X-form
111111 ..... 01000 ..... 11001 00100/	I	...X	BFP128		xsnabsqp	v3.0			609	VSX Scalar Negative Absolute Quad-Precision X-form
111111 ..... 10000 ..... 11001 00100/	I	...X	BFP128		xsnegqp	v3.0			610	VSX Scalar Negate Quad-Precision X-form
111111 .... 01110 00100.	I	...X	BFP128		xsnmaddqp[o]	v3.0			673	VSX Scalar Negative Multiply-Add Quad-Precision [using round to Odd] X-form
111111 .... 01111 00100.	I	...X	BFP128		xsnmsubqp[o]	v3.0			682	VSX Scalar Negative Multiply-Subtract Quad-Precision [using round to Odd] X-form
111111 ..... ////. ....000 00101.	I	...X	BFP128		xsrqpi[x]	v3.0			806	VSX Scalar Round to Quad-Precision Integer [with Inexact] Z23-form
111111 ..... ////. 001 00101/	I	...X	BFP128		xsrqpxp	v3.0			785	VSX Scalar Round Quad-Precision to Double-Extended-Precision Z23-form
111111 ..... 11011 ..... 11001 00100.	I	...X	BFP128		xssqrtqp[o]	v3.0			638	VSX Scalar Square Root Quad-Precision [using round to Odd] X-form
111111 .... 10000 00100.	I	...X	BFP128		xssubqp[o]	v3.0			644	VSX Scalar Subtract Quad-Precision [using round to Odd] X-form
111100 .... 10110 1010./	I	..XX			xststdcdp	v3.0			866	VSX Scalar Test Data Class Double-Precision XX2-form
111111 .... 10110 00100/	I	...X	BFP128		xststdcqp	v3.0			867	VSX Scalar Test Data Class Quad-Precision X-form
111100 .... 10010 1010./	I	..XX			xststdcsp	v3.0			868	VSX Scalar Test Data Class Single-Precision XX2-form
111100 ..... 00000 ..... 10101 1011./	I	..XX			xsxexpdp	v3.0			869	VSX Scalar Extract Exponent Double-Precision XX2-form
111111 ..... 00010 ..... 11001 00100/	I	...X	BFP128		xsxexpqp	v3.0			869	VSX Scalar Extract Exponent Quad-Precision X-form
111100 ..... 00001 ..... 10101 1011./	I	..XX			xsxsigdp	v3.0			870	VSX Scalar Extract Significand Double-Precision XX2-form

Table F.1: Power ISA Instruction Set Sorted by Version (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
111111 ..... 10010 ..... 11001 00100/	I	...X	BFP128		xsxsigqp	v3.0			870	VSX Scalar Extract Significand Quad-Precision X-form
111100 ..... 11000 ..... 11101 1011..	I	..XX			xvcvhpsp	v3.0			799	VSX Vector Convert Half-Precision to Single-Precision format XX2-form
111100 ..... 11001 ..... 11101 1011..	I	..XX			xvcvsphp	v3.0			794	VSX Vector Convert with round Single-Precision to Half-Precision format XX2-form
111100 .... 11111 000...	I	..XX			xviexpdp	v3.0			871	VSX Vector Insert Exponent Double-Precision XX3-form
111100 .... 11011 000...	I	..XX			xviexpsp	v3.0			871	VSX Vector Insert Exponent Single-Precision XX3-form
111100 .... 1111. 101...	I	..XX			xvtstdcdp	v3.0			872	VSX Vector Test Data Class Double-Precision XX2-form
111100 .... 1101. 101...	I	..XX			xvtstdcsp	v3.0			873	VSX Vector Test Data Class Single-Precision XX2-form
111100 ..... 00000 ..... 11101 1011..	I	..XX			xvxexpdp	v3.0			874	VSX Vector Extract Exponent Double-Precision XX2-form
111100 ..... 01000 ..... 11101 1011..	I	..XX			xvxexpsp	v3.0			874	VSX Vector Extract Exponent Single-Precision XX2-form
111100 ..... 00001 ..... 11101 1011..	I	..XX			xvxsigdp	v3.0			875	VSX Vector Extract Significand Double-Precision XX2-form
111100 ..... 01001 ..... 11101 1011..	I	..XX			xvxsigsp	v3.0			875	VSX Vector Extract Significand Single-Precision XX2-form
111100 ..... 10111 ..... 11101 1011..	I	..XX			xxbrd	v3.0			914	VSX Vector Byte-Reverse Doubleword XX2-form
111100 ..... 00111 ..... 11101 1011..	I	..XX			xxbrh	v3.0			915	VSX Vector Byte-Reverse Halfword XX2-form
111100 ..... 11111 ..... 11101 1011..	I	..XX			xxbrq	v3.0			915	VSX Vector Byte-Reverse Quadword XX2-form
111100 ..... 01111 ..... 11101 1011..	I	..XX			xxbrw	v3.0			916	VSX Vector Byte-Reverse Word XX2-form
111100 ..... /.... ..... 01010 0101..	I	..XX			xxextractuw	v3.0			917	VSX Vector Extract Unsigned Word XX2-form
111100 ..... /.... ..... 01011 0101..	I	..XX			xxinsertw	v3.0			917	VSX Vector Insert Word XX2-form
111100 .... 00011 010...	I	..XX			xxperm	v3.0			923	VSX Vector Permute XX3-form
111100 .... 00111 010...	I	..XX			xxpermr	v3.0			923	VSX Vector Permute Right-indexed XX3-form
111100 ..... 00... ..... 01011 01000.	I	..XX			xxspltib	v3.0			919	VSX Vector Splat Immediate Byte X-form
000100 .... 1.000 000001	I	..XX			bcdadd.	v2.07			466	Decimal Add Modulo VX-form
000100 .... 1.001 000001	I	..XX			bcdsub.	v2.07			466	Decimal Subtract Modulo VX-form
010011 .... ///.. 10001 10000.	I	XXXX			bctar[l]	v2.07			42	Branch Conditional to Branch Target Address Register XL-form
011111 ///// ///// ///// 01101 01110/	I	...X	BHRB		clrbhrb	v2.07			1041	Clear BHRB X-form
111111 .... 11110 00110/	I	.XXX			fmrgew	v2.07			161	Floating Merge Even Word X-form
111111 .... 11010 00110/	I	.XXX			fmrgow	v2.07			161	Floating Merge Odd Word X-form
011111 /.... .... 00000 10110/	II	..XX			icbt	v2.07			991	Instruction Cache Block Touch X-form
011111 .... 01000 10100.	I	..XX			lqarx	v2.07			1023	Load Quadword And Reserve Indexed X-form
011111 .... 00010 01100.	I	..XX			lxsiwax	v2.07			565	Load VSX Scalar as Integer Word Algebraic Indexed X-form
011111 .... 00000 01100.	I	..XX			lxsiwzx	v2.07			566	Load VSX Scalar as Integer Word & Zero Indexed X-form
011111 .... 10000 01100.	I	..XX			lxsspx	v2.07			568	Load VSX Scalar Single-Precision Indexed X-form
011111 .... 01001 01110/	I	...X	BHRB		mfbhrbe	v2.07			1041	Move From Branch History Rolling Buffer Entry XFX-form
011111 .... ///// 00001 10011.	I	..XX			mfvsrd	v2.07			122	Move From VSR Doubleword X-form
011111 .... ///// 00011 10011.	I	..XX			mfvsrwz	v2.07			123	Move From VSR Word and Zero X-form
011111 ///// ///// 00111 01110/	III	..XX			msgclr	v2.07	HV		1275	Message Clear X-form
011111 ///// ///// 00101 01110/	III	..XX			msgclrp	v2.07	P		1276	Message Clear Privileged X-form
011111 ///// ///// 00110 01110/	III	..XX			msgsnd	v2.07	HV		1275	Message Send X-form
011111 ///// ///// 00100 01110/	III	..XX			msgsndp	v2.07	P		1276	Message Send Privileged X-form
011111 .... ///// 00101 10011.	I	..XX			mtvsrd	v2.07			123	Move To VSR Doubleword X-form
011111 .... ///// 00110 10011.	I	..XX			mtvsrwa	v2.07			124	Move To VSR Word Algebraic X-form
011111 .... ///// 00111 10011.	I	..XX			mtvsrwz	v2.07			124	Move To VSR Word and Zero X-form
010011 ///// ///// ////. 00100 10010/	I	...X	EBB		rfebb	v2.07			1037	Return from Event-Based Branch XL-form
011111 .... 00101 101101	I	..XX			stqcx.	v2.07			1024	Store Quadword Conditional Indexed X-form

Table F.1: Power ISA Instruction Set Sorted by Version (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
011111 .... 00100 01100.	I	..XX			stxsiwx	v2.07			572	Store VSX Scalar as Integer Word Indexed X-form
011111 .... 10100 01100.	I	..XX			stxsspx	v2.07			574	Store VSX Scalar Single-Precision Indexed X-form
000100 .... 00101 000000	I	..XX			vaddcuq	v2.07			320	Vector Add & write Carry-out Unsigned Quadword VX-form
000100 111101	I	..XX			vaddecuq	v2.07			320	Vector Add Extended & write Carry-out Unsigned Quadword VA-form
000100 111100	I	..XX			vaddeuqm	v2.07			319	Vector Add Extended Unsigned Quadword Modulo VA-form
000100 .... 00011 000000	I	..XX			vaddudm	v2.07			316	Vector Add Unsigned Doubleword Modulo VX-form
000100 .... 00100 000000	I	..XX			vadduqm	v2.07			319	Vector Add Unsigned Quadword Modulo VX-form
000100 .... 10101 001100	I	..XX			vbpermq	v2.07			450	Vector Bit Permute Quadword VX-form
000100 .... 10100 001000	I	..XX			vcipher	v2.07			424	Vector AES Cipher VX-form
000100 .... 10100 001001	I	..XX			vcipherlast	v2.07			424	Vector AES Cipher Last VX-form
000100 ..... ///// 11100 000010	I	..XX			vclzb	v2.07			435	Vector Count Leading Zeros Byte VX-form
000100 ..... ///// 11111 000010	I	..XX			vclzd	v2.07			437	Vector Count Leading Zeros Doubleword VX-form
000100 ..... ///// 11101 000010	I	..XX			vclzh	v2.07			435	Vector Count Leading Zeros Halfword VX-form
000100 ..... ///// 11110 000010	I	..XX			vclzw	v2.07			436	Vector Count Leading Zeros Word VX-form
000100 .... .0011 000111	I	..XX			vcmpequd[.]	v2.07			381	Vector Compare Equal Unsigned Doubleword VC-form
000100 .... .1111 000111	I	..XX			vcmpgtsd[.]	v2.07			386	Vector Compare Greater Than Signed Doubleword VC-form
000100 .... .1011 000111	I	..XX			vcmpgtud[.]	v2.07			386	Vector Compare Greater Than Unsigned Doubleword VC-form
000100 .... 11010 000100	I	..XX			veqv	v2.07			393	Vector Logical Equivalence VX-form
000100 ..... ///// 10100 001100	I	..XX			vgbbd	v2.07			433	Vector Gather Bits by Bytes by Doubleword VX-form
000100 .... 00111 000010	I	..XX			vmaxsd	v2.07			373	Vector Maximum Signed Doubleword VX-form
000100 .... 00011 000010	I	..XX			vmaxud	v2.07			373	Vector Maximum Unsigned Doubleword VX-form
000100 .... 01111 000010	I	..XX			vminsd	v2.07			377	Vector Minimum Signed Doubleword VX-form
000100 .... 01011 000010	I	..XX			vminud	v2.07			377	Vector Minimum Unsigned Doubleword VX-form
000100 .... 11110 001100	I	..XX			vmrgew	v2.07			282	Vector Merge Even Word VX-form
000100 .... 11010 001100	I	..XX			vmrgow	v2.07			282	Vector Merge Odd Word VX-form
000100 .... 01110 001000	I	..XX			vmulesw	v2.07			333	Vector Multiply Even Signed Word VX-form
000100 .... 01010 001000	I	..XX			vmuleuw	v2.07			334	Vector Multiply Even Unsigned Word VX-form
000100 .... 00110 001000	I	..XX			vmulosw	v2.07			333	Vector Multiply Odd Signed Word VX-form
000100 .... 00010 001000	I	..XX			vmulouw	v2.07			334	Vector Multiply Odd Unsigned Word VX-form
000100 .... 00010 001001	I	..XX			vmuluwm	v2.07			337	Vector Multiply Unsigned Word Modulo VX-form
000100 .... 10110 000100	I	..XX			vnand	v2.07			393	Vector Logical NAND VX-form
000100 .... 10101 001000	I	..XX			vncipher	v2.07			425	Vector AES Inverse Cipher VX-form
000100 .... 10101 001001	I	..XX			vncipherlast	v2.07			425	Vector AES Inverse Cipher Last VX-form
000100 .... 10101 000100	I	..XX			vorc	v2.07			393	Vector Logical OR with Complement VX-form
000100 101101	I	..XX			vpermxor	v2.07			432	Vector Permute & Exclusive-OR VA-form
000100 .... 10111 001110	I	..XX			vpksdss	v2.07			271	Vector Pack Signed Doubleword Signed Saturate VX-form
000100 .... 10101 001110	I	..XX			vpksdus	v2.07			271	Vector Pack Signed Doubleword Unsigned Saturate VX-form
000100 .... 10001 001110	I	..XX			vpkudum	v2.07			274	Vector Pack Unsigned Doubleword Unsigned Modulo VX-form
000100 .... 10011 001110	I	..XX			vpkudus	v2.07			274	Vector Pack Unsigned Doubleword Unsigned Saturate VX-form
000100 .... 10000 001000	I	..XX			vpmsumb	v2.07			430	Vector Polynomial Multiply-Sum Byte VX-form
000100 .... 10011 001000	I	..XX			vpmsumd	v2.07			431	Vector Polynomial Multiply-Sum Doubleword VX-form

Table F.1: Power ISA Instruction Set Sorted by Version (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
000100 .... 10001 001000	I	..XX			vpmsumh	v2.07			430	Vector Polynomial Multiply-Sum Halfword VX-form
000100 .... 10010 001000	I	..XX			vpmsumw	v2.07			431	Vector Polynomial Multiply-Sum Word VX-form
000100 ..... ///// 11100 000011	I	..XX			vpopcntb	v2.07			445	Vector Population Count Byte VX-form
000100 ..... ///// 11111 000011	I	..XX			vpopcntd	v2.07			446	Vector Population Count Doubleword VX-form
000100 ..... ///// 11101 000011	I	..XX			vpopcnth	v2.07			445	Vector Population Count Halfword VX-form
000100 ..... ///// 11110 000011	I	..XX			vpopcntw	v2.07			446	Vector Population Count Word VX-form
000100 .... 00011 000100	I	..XX			vrld	v2.07			396	Vector Rotate Left Doubleword VX-form
000100 .... ///// 10111 001000	I	..XX			vsbox	v2.07			426	Vector AES SubBytes VX-form
000100 .... 11011 000010	I	..XX			vshasigmad	v2.07			427	Vector SHA-512 Sigma Doubleword VX-form
000100 .... 11010 000010	I	..XX			vshasigmaw	v2.07			428	Vector SHA-256 Sigma Word VX-form
000100 .... 10111 000100	I	..XX			vsld	v2.07			403	Vector Shift Left Doubleword VX-form
000100 .... 01111 000100	I	..XX			vsrad	v2.07			409	Vector Shift Right Algebraic Doubleword VX-form
000100 .... 11011 000100	I	..XX			vsrd	v2.07			406	Vector Shift Right Doubleword VX-form
000100 .... 10101 000000	I	..XX			vsubcuq	v2.07			328	Vector Subtract & write Carry-out Unsigned Quadword VX-form
000100 111111	I	..XX			vsubecuq	v2.07			328	Vector Subtract Extended & write Carry-out Unsigned Quadword VA-form
000100 111110	I	..XX			vsubeuqm	v2.07			327	Vector Subtract Extended Unsigned Quadword Modulo VA-form
000100 .... 10011 000000	I	..XX			vsubudm	v2.07			324	Vector Subtract Unsigned Doubleword Modulo VX-form
000100 .... 10100 000000	I	..XX			vsubuqm	v2.07			327	Vector Subtract Unsigned Quadword Modulo VX-form
000100 ..... ///// 11001 001110	I	..XX			vupkhsw	v2.07			277	Vector Unpack High Signed Word VX-form
000100 ..... ///// 11011 001110	I	..XX			vupklsw	v2.07			277	Vector Unpack Low Signed Word VX-form
111100 .... 00000 000...	I	..XX			xsaddsp	v2.07			622	VSX Scalar Add Single-Precision XX3-form
111100 ..... ///// ..... 10000 1011..	I	..XX			xscvdpspn	v2.07			790	VSX Scalar Convert Scalar Single-Precision to Vector Single-Precision format Non-signalling XX2-form
111100 ..... ///// ..... 10100 1011..	I	..XX			xscvspdpn	v2.07			798	VSX Scalar Convert Single-Precision to Double-Precision format Non-signalling XX2-form
111100 ..... ///// ..... 10011 1000..	I	..XX			xscvsxdsp	v2.07			855	VSX Scalar Convert with round Signed Doubleword to Single-Precision format XX2-form
111100 ..... ///// ..... 10010 1000..	I	..XX			xscvuxdsp	v2.07			855	VSX Scalar Convert with round Unsigned Doubleword to Single-Precision format XX2-form
111100 .... 00011 000...	I	..XX			xsdivsp	v2.07			628	VSX Scalar Divide Single-Precision XX3-form
111100 .... 00000 001...	I	..XX			xsmaddasp	v2.07			651	VSX Scalar Multiply-Add Type-A Single-Precision XX3-form
111100 .... 00001 001...	I	..XX			xsmaddmsp	v2.07			651	VSX Scalar Multiply-Add Type-M Single-Precision XX3-form
111100 .... 00010 001...	I	..XX			xsmsubasp	v2.07			660	VSX Scalar Multiply-Subtract Type-A Single-Precision XX3-form
111100 .... 00011 001...	I	..XX			xsmsubmsp	v2.07			660	VSX Scalar Multiply-Subtract Type-M Single-Precision XX3-form
111100 .... 00010 000...	I	..XX			xsmulsp	v2.07			634	VSX Scalar Multiply Single-Precision XX3-form
111100 .... 10000 001...	I	..XX			xsnmaddasp	v2.07			670	VSX Scalar Negative Multiply-Add Type-A Single-Precision XX3-form
111100 .... 10001 001...	I	..XX			xsnmaddmsp	v2.07			670	VSX Scalar Negative Multiply-Add Type-M Single-Precision XX3-form
111100 .... 10010 001...	I	..XX			xsnmsubasp	v2.07			679	VSX Scalar Negative Multiply-Subtract Type-A Single-Precision XX3-form

Table F.1: Power ISA Instruction Set Sorted by Version (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
111100 .... 10011 001...	I	..XX			xsnmsubmsp	v2.07			679	VSX Scalar Negative Multiply-Subtract Type-M Single-Precision XX3-form
111100 ..... ///// ..... 00001 1010..	I	..XX			xsresp	v2.07			686	VSX Scalar Reciprocal Estimate Single-Precision XX2-form
111100 ..... ///// ..... 10001 1001..	I	..XX			xsrsp	v2.07			787	VSX Scalar Round to Single-Precision XX2-form
111100 ..... ///// ..... 00000 1010..	I	..XX			xsrsqrtesp	v2.07			688	VSX Scalar Reciprocal Square Root Estimate Single-Precision XX2-form
111100 ..... ///// ..... 00000 1011..	I	..XX			xssqrtsp	v2.07			640	VSX Scalar Square Root Single-Precision XX2-form
111100 .... 00001 000...	I	..XX			xssubsp	v2.07			646	VSX Scalar Subtract Single-Precision XX3-form
111100 .... 10111 010...	I	..XX			xxleqv	v2.07			907	VSX Vector Logical Equivalence XX3-form
111100 .... 10110 010...	I	..XX			xxlnand	v2.07			907	VSX Vector Logical NAND XX3-form
111100 .... 10101 010...	I	..XX			xxlorc	v2.07			908	VSX Vector Logical OR with Complement XX3-form
011111 .... /0010 01010/	I	XXXX			addg6s	v2.06			117	Add and Generate Sixes XO-form
011111 .... 00111 11100/	I	..XX			bpermd	v2.06			105	Bit Permute Doubleword X-form
011111 .... ///// 01001 11010/	I	XXXX			cbcdtd	v2.06			117	Convert Binary Coded Decimal To Declets X-form
011111 .... ///// 01000 11010/	I	XXXX			cdtbcd	v2.06			117	Convert Declets To Binary Coded Decimal X-form
111011 ..... ///// ..... 11001 00010.	I	...X	DFP		dcffix[.]	v2.06			227	DFP Convert From Fixed X-form
011111 .... 01101 01001.	I	..XX			divde[.]	v2.06		SR	88	Divide Doubleword Extended XO-form
011111 .... 11101 01001.	I	..XX			divdeo[.]	v2.06		SR	88	Divide Doubleword Extended & record OV XO-form
011111 .... 01100 01001.	I	..XX			divdeu[.]	v2.06		SR	88	Divide Doubleword Extended Unsigned XO-form
011111 .... 11100 01001.	I	..XX			divdeuo[.]	v2.06		SR	88	Divide Doubleword Extended Unsigned & record OV XO-form
011111 .... 01101 01011.	I	XXXX			divwe[.]	v2.06		SR	81	Divide Word Extended XO-form
011111 .... 11101 01011.	I	..XX			divweo[.]	v2.06		SR	81	Divide Word Extended & record OV XO-form
011111 .... 01100 01011.	I	XXXX			divweu[.]	v2.06		SR	81	Divide Word Extended Unsigned XO-form
011111 .... 11100 01011.	I	..XX			divweuo[.]	v2.06		SR	81	Divide Word Extended Unsigned & record OV XO-form
111011 ..... ///// ..... 11010 01110.	I	.XXX			fcfids[.]	v2.06			176	Floating Convert with round Signed Doubleword to Single-Precision format X-form
111111 ..... ///// ..... 11110 01110.	I	.XXX			fcfidu[.]	v2.06			175	Floating Convert with round Unsigned Doubleword to Double-Precision format X-form
111011 ..... ///// ..... 11110 01110.	I	.XXX			fcfidus[.]	v2.06			176	Floating Convert with round Unsigned Doubleword to Single-Precision format X-form
111111 ..... ///// ..... 11101 01110.	I	.XXX			fctidu[.]	v2.06			172	Floating Convert with round Double-Precision To Unsigned Doubleword format X-form
111111 ..... ///// ..... 11101 01111.	I	.XXX			fctiduz[.]	v2.06			172	Floating Convert with truncate Double-Precision To Unsigned Doubleword format X-form
111111 ..... ///// ..... 00100 01110.	I	.XXX			fctiwu[.]	v2.06			174	Floating Convert with round Double-Precision To Unsigned Word format X-form
111111 ..... ///// ..... 00100 01111.	I	.XXX			fctiwuz[.]	v2.06			174	Floating Convert with truncate Double-Precision To Unsigned Word format X-form
111111 ...// 00100 00000/	I	.XXX			ftdiv	v2.06			167	Floating Test for software Divide X-form
111111 ...// ///// 00101 00000/	I	.XXX			ftsqrt	v2.06			167	Floating Test for software Square Root X-form
011111 .... 00001 10100.	II	XXXX			lbarx	v2.06			1016	Load Byte And Reserve Indexed X-form
011111 .... 10000 10100/	I	..XX			ldbrx	v2.06			67	Load Doubleword Byte-Reverse Indexed X-form
011111 .... 11011 10111/	I	.XXX			lfiwzx	v2.06			153	Load Floating-Point as Integer Word & Zero Indexed X-form
011111 .... 00011 10100.	II	XXXX			lharx	v2.06			1016	Load Halfword And Reserve Indexed X-form
011111 .... 10010 01100.	I	..XX			lxsdx	v2.06			563	Load VSX Scalar Doubleword Indexed X-form
011111 .... 11010 01100.	I	..XX			lxvd2x	v2.06			577	Load VSX Vector Doubleword*2 Indexed X-form

Table F.1: Power ISA Instruction Set Sorted by Version (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
011111 .... 01010 01100.	I	..XX			lxvdsx	v2.06			582	Load VSX Vector Doubleword & Splat Indexed X-form
011111 .... 11000 01100.	I	..XX			lxvw4x	v2.06			579	Load VSX Vector Word*4 Indexed X-form
011111 .... ///// 01111 11010/	I	..XX			popcntd	v2.06			103	Population Count Doubleword X-form
011111 .... ///// 01011 11010/	I	XXXX			popcntw	v2.06			102	Population Count Words X-form
011111 .... 10101 101101	II	XXXX			stbcx.	v2.06			1018	Store Byte Conditional Indexed X-form
011111 .... 10100 10100/	I	..XX			stdbrx	v2.06			67	Store Doubleword Byte-Reverse Indexed X-form
011111 .... 10110 101101	II	XXXX			sthcx.	v2.06			1019	Store Halfword Conditional Indexed X-form
011111 .... 10110 01100.	I	..XX			stxsdx	v2.06			570	Store VSX Scalar Doubleword Indexed X-form
011111 .... 11110 01100.	I	..XX			stxvd2x	v2.06			593	Store VSX Vector Doubleword*2 Indexed X-form
011111 .... 11100 01100.	I	..XX			stxvw4x	v2.06			595	Store VSX Vector Word*4 Indexed X-form
111100 ..... ///// ..... 10101 1001..	I	..XX			xsabsdp	v2.06			607	VSX Scalar Absolute Double-Precision XX2-form
111100 .... 00100 000...	I	..XX			xsadddp	v2.06			615	VSX Scalar Add Double-Precision XX3-form
111100 ...// .... 00101 011../	I	..XX			xscmpodp	v2.06			743	VSX Scalar Compare Ordered Double-Precision XX3-form
111100 ...// .... 00100 011../	I	..XX			xscmpudp	v2.06			746	VSX Scalar Compare Unordered Double-Precision XX3-form
111100 .... 10110 000...	I	..XX			xscpsgndp	v2.06			608	VSX Scalar Copy Sign Double-Precision XX3-form
111100 ..... ///// ..... 10000 1001..	I	..XX			xscvdpsp	v2.06			789	VSX Scalar Convert with round Double-Precision to Single-Precision format XX2-form
111100 ..... ///// ..... 10101 1000..	I	..XX			xscvdpsxds	v2.06			816	VSX Scalar Convert with round to zero Double-Precision to Signed Doubleword format XX2-form
111100 ..... ///// ..... 00101 1000..	I	..XX			xscvdpsxws	v2.06			818	VSX Scalar Convert with round to zero Double-Precision to Signed Word format XX2-form
111100 ..... ///// ..... 10100 1000..	I	..XX			xscvdpuxds	v2.06			820	VSX Scalar Convert with round to zero Double-Precision to Unsigned Doubleword format XX2-form
111100 ..... ///// ..... 00100 1000..	I	..XX			xscvdpuxws	v2.06			822	VSX Scalar Convert with round to zero Double-Precision to Unsigned Word format XX2-form
111100 ..... ///// ..... 10100 1001..	I	..XX			xscvspdp	v2.06			797	VSX Scalar Convert Single-Precision to Double-Precision format XX2-form
111100 ..... ///// ..... 10111 1000..	I	..XX			xscvsxddp	v2.06			854	VSX Scalar Convert with round Signed Doubleword to Double-Precision format XX2-form
111100 ..... ///// ..... 10110 1000..	I	..XX			xscvuxddp	v2.06			854	VSX Scalar Convert with round Unsigned Doubleword to Double-Precision format XX2-form
111100 .... 00111 000...	I	..XX			xsdivdp	v2.06			624	VSX Scalar Divide Double-Precision XX3-form
111100 .... 00100 001...	I	..XX			xsmaddadp	v2.06			648	VSX Scalar Multiply-Add Type-A Double-Precision XX3-form
111100 .... 00101 001...	I	..XX			xsmaddmdp	v2.06			648	VSX Scalar Multiply-Add Type-M Double-Precision XX3-form
111100 .... 10100 000...	I	..XX			xsmaxdp	v2.06			759	VSX Scalar Maximum Double-Precision XX3-form
111100 .... 10101 000...	I	..XX			xsmindp	v2.06			767	VSX Scalar Minimum Double-Precision XX3-form
111100 .... 00110 001...	I	..XX			xsmsubadp	v2.06			657	VSX Scalar Multiply-Subtract Type-A Double-Precision XX3-form
111100 .... 00111 001...	I	..XX			xsmsubmdp	v2.06			657	VSX Scalar Multiply-Subtract Type-M Double-Precision XX3-form
111100 .... 00110 000...	I	..XX			xsmuldp	v2.06			630	VSX Scalar Multiply Double-Precision XX3-form
111100 ..... ///// ..... 10110 1001..	I	..XX			xsnabsdp	v2.06			609	VSX Scalar Negative Absolute Double-Precision XX2-form
111100 ..... ///// ..... 10111 1001..	I	..XX			xsnegdp	v2.06			610	VSX Scalar Negate Double-Precision XX2-form
111100 .... 10100 001...	I	..XX			xsnmaddadp	v2.06			666	VSX Scalar Negative Multiply-Add Type-A Double-Precision XX3-form

Table F.1: Power ISA Instruction Set Sorted by Version (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
111100 .... 10101 001...	I	..XX			xsnmaddmdp	v2.06			666	VSX Scalar Negative Multiply-Add Type-M Double-Precision XX3-form
111100 .... 10110 001...	I	..XX			xsnmsubadp	v2.06			676	VSX Scalar Negative Multiply-Subtract Type-A Double-Precision XX3-form
111100 .... 10111 001...	I	..XX			xsnmsubmdp	v2.06			676	VSX Scalar Negative Multiply-Subtract Type-M Double-Precision XX3-form
111100 ..... ///// ..... 00100 1001..	I	..XX			xsrdpi	v2.06			801	VSX Scalar Round to Double-Precision Integer using round to Nearest Away XX2-form
111100 ..... ///// ..... 00110 1011..	I	..XX			xsrdpic	v2.06			802	VSX Scalar Round to Double-Precision Integer exact using Current rounding mode XX2-form
111100 ..... ///// ..... 00111 1001..	I	..XX			xsrdpim	v2.06			803	VSX Scalar Round to Double-Precision Integer using round toward -Infinity XX2-form
111100 ..... ///// ..... 00110 1001..	I	..XX			xsrdpip	v2.06			804	VSX Scalar Round to Double-Precision Integer using round toward +Infinity XX2-form
111100 ..... ///// ..... 00101 1001..	I	..XX			xsrdpiz	v2.06			805	VSX Scalar Round to Double-Precision Integer using round toward Zero XX2-form
111100 ..... ///// ..... 00101 1010..	I	..XX			xsredp	v2.06			685	VSX Scalar Reciprocal Estimate Double-Precision XX2-form
111100 ..... ///// ..... 00100 1010..	I	..XX			xsrsqrtedp	v2.06			687	VSX Scalar Reciprocal Square Root Estimate Double-Precision XX2-form
111100 ..... ///// ..... 00100 1011..	I	..XX			xssqrtdp	v2.06			636	VSX Scalar Square Root Double-Precision XX2-form
111100 .... 00101 000...	I	..XX			xssubdp	v2.06			642	VSX Scalar Subtract Double-Precision XX3-form
111100 .../// .... 00111 101../	I	..XX			xstdivdp	v2.06			689	VSX Scalar Test for software Divide Double-Precision XX3-form
111100 .../// ///// 00110 1010./	I	..XX			xstsqrtdp	v2.06			690	VSX Scalar Test for software Square Root Double-Precision XX2-form
111100 ..... ///// ..... 11101 1001..	I	..XX			xvabsdp	v2.06			611	VSX Vector Absolute Double-Precision XX2-form
111100 ..... ///// ..... 11001 1001..	I	..XX			xvabssp	v2.06			611	VSX Vector Absolute Single-Precision XX2-form
111100 .... 01100 000...	I	..XX			xvadddp	v2.06			691	VSX Vector Add Double-Precision XX3-form
111100 .... 01000 000...	I	..XX			xvaddsp	v2.06			694	VSX Vector Add Single-Precision XX3-form
111100 .... .1100 011...	I	..XX			xvcmpeqdp[.]	v2.06			771	VSX Vector Compare Equal To Double-Precision XX3-form
111100 .... .1000 011...	I	..XX			xvcmpeqsp[.]	v2.06			772	VSX Vector Compare Equal To Single-Precision XX3-form
111100 .... .1110 011...	I	..XX			xvcmpgedp[.]	v2.06			773	VSX Vector Compare Greater Than or Equal To Double-Precision XX3-form
111100 .... .1010 011...	I	..XX			xvcmpgesp[.]	v2.06			774	VSX Vector Compare Greater Than or Equal To Single-Precision XX3-form
111100 .... .1101 011...	I	..XX			xvcmpgtdp[.]	v2.06			775	VSX Vector Compare Greater Than Double-Precision XX3-form
111100 .... .1001 011...	I	..XX			xvcmpgtsp[.]	v2.06			776	VSX Vector Compare Greater Than Single-Precision XX3-form
111100 .... 11110 000...	I	..XX			xvcpsgndp	v2.06			612	VSX Vector Copy Sign Double-Precision XX3-form
111100 .... 11010 000...	I	..XX			xvcpsgnsp	v2.06			612	VSX Vector Copy Sign Single-Precision XX3-form
111100 ..... ///// ..... 11000 1001..	I	..XX			xvcvdpsp	v2.06			793	VSX Vector Convert with round Double-Precision to Single-Precision format XX2-form
111100 ..... ///// ..... 11101 1000..	I	..XX			xvcvdpsxds	v2.06			836	VSX Vector Convert with round to zero Double-Precision to Signed Doubleword format XX2-form
111100 ..... ///// ..... 01101 1000..	I	..XX			xvcvdpsxws	v2.06			838	VSX Vector Convert with round to zero Double-Precision to Signed Word format XX2-form

Table F.1: Power ISA Instruction Set Sorted by Version (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
111100 ..... ///// ..... 11100 1000..	I	..XX			xvcvdpuxds	v2.06			840	VSX Vector Convert with round to zero Double-Precision to Unsigned Doubleword format XX2-form
111100 ..... ///// ..... 01100 1000..	I	..XX			xvcvdpuxws	v2.06			842	VSX Vector Convert with round to zero Double-Precision to Unsigned Word format XX2-form
111100 ..... ///// ..... 11100 1001..	I	..XX			xvcvspdp	v2.06			800	VSX Vector Convert Single-Precision to Double-Precision format XX2-form
111100 ..... ///// ..... 11001 1000..	I	..XX			xvcvspsxds	v2.06			844	VSX Vector Convert with round to zero Single-Precision to Signed Doubleword format XX2-form
111100 ..... ///// ..... 01001 1000..	I	..XX			xvcvspsxws	v2.06			846	VSX Vector Convert with round to zero Single-Precision to Signed Word format XX2-form
111100 ..... ///// ..... 11000 1000..	I	..XX			xvcvspuxds	v2.06			848	VSX Vector Convert with round to zero Single-Precision to Unsigned Doubleword format XX2-form
111100 ..... ///// ..... 01000 1000..	I	..XX			xvcvspuxws	v2.06			850	VSX Vector Convert with round to zero Single-Precision to Unsigned Word format XX2-form
111100 ..... ///// ..... 11111 1000..	I	..XX			xvcvsxddp	v2.06			857	VSX Vector Convert with round Signed Doubleword to Double-Precision format XX2-form
111100 ..... ///// ..... 11011 1000..	I	..XX			xvcvsxdsp	v2.06			859	VSX Vector Convert with round Signed Doubleword to Single-Precision format XX2-form
111100 ..... ///// ..... 01111 1000..	I	..XX			xvcvsxwdp	v2.06			858	VSX Vector Convert Signed Word to Double-Precision format XX2-form
111100 ..... ///// ..... 01011 1000..	I	..XX			xvcvsxwsp	v2.06			861	VSX Vector Convert with round Signed Word to Single-Precision format XX2-form
111100 ..... ///// ..... 11110 1000..	I	..XX			xvcvuxddp	v2.06			857	VSX Vector Convert with round Unsigned Doubleword to Double-Precision format XX2-form
111100 ..... ///// ..... 11010 1000..	I	..XX			xvcvuxdsp	v2.06			859	VSX Vector Convert with round Unsigned Doubleword to Single-Precision format XX2-form
111100 ..... ///// ..... 01110 1000..	I	..XX			xvcvuxwdp	v2.06			858	VSX Vector Convert Unsigned Word to Double-Precision format XX2-form
111100 ..... ///// ..... 01010 1000..	I	..XX			xvcvuxwsp	v2.06			861	VSX Vector Convert with round Unsigned Word to Single-Precision format XX2-form
111100 .... 01111 000...	I	..XX			xvdivdp	v2.06			696	VSX Vector Divide Double-Precision XX3-form
111100 .... 01011 000...	I	..XX			xvdivsp	v2.06			698	VSX Vector Divide Single-Precision XX3-form
111100 .... 01100 001...	I	..XX			xvmaddadp	v2.06			710	VSX Vector Multiply-Add Type-A Double-Precision XX3-form
111100 .... 01000 001...	I	..XX			xvmaddasp	v2.06			713	VSX Vector Multiply-Add Type-A Single-Precision XX3-form
111100 .... 01101 001...	I	..XX			xvmaddmdp	v2.06			710	VSX Vector Multiply-Add Type-M Double-Precision XX3-form
111100 .... 01001 001...	I	..XX			xvmaddmsp	v2.06			713	VSX Vector Multiply-Add Type-M Single-Precision XX3-form
111100 .... 11100 000...	I	..XX			xvmaxdp	v2.06			777	VSX Vector Maximum Double-Precision XX3-form
111100 .... 11000 000...	I	..XX			xvmaxsp	v2.06			779	VSX Vector Maximum Single-Precision XX3-form
111100 .... 11101 000...	I	..XX			xvmindp	v2.06			781	VSX Vector Minimum Double-Precision XX3-form
111100 .... 11001 000...	I	..XX			xvminsp	v2.06			783	VSX Vector Minimum Single-Precision XX3-form
111100 .... 01110 001...	I	..XX			xvmsubadp	v2.06			716	VSX Vector Multiply-Subtract Type-A Double-Precision XX3-form
111100 .... 01010 001...	I	..XX			xvmsubasp	v2.06			719	VSX Vector Multiply-Subtract Type-A Single-Precision XX3-form

Table F.1: Power ISA Instruction Set Sorted by Version (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
111100 .... 01111 001...	I	..XX			xvmsubmdp	v2.06			716	VSX Vector Multiply-Subtract Type-M Double-Precision XX3-form
111100 .... 01011 001...	I	..XX			xvmsubmsp	v2.06			719	VSX Vector Multiply-Subtract Type-M Single-Precision XX3-form
111100 .... 01110 000...	I	..XX			xvmuldp	v2.06			700	VSX Vector Multiply Double-Precision XX3-form
111100 .... 01010 000...	I	..XX			xvmulsp	v2.06			702	VSX Vector Multiply Single-Precision XX3-form
111100 ..... ///// ..... 11110 1001..	I	..XX			xvnabsdp	v2.06			613	VSX Vector Negative Absolute Double-Precision XX2-form
111100 ..... ///// ..... 11010 1001..	I	..XX			xvnabssp	v2.06			613	VSX Vector Negative Absolute Single-Precision XX2-form
111100 ..... ///// ..... 11111 1001..	I	..XX			xvnegdp	v2.06			614	VSX Vector Negate Double-Precision XX2-form
111100 ..... ///// ..... 11011 1001..	I	..XX			xvnegsp	v2.06			614	VSX Vector Negate Single-Precision XX2-form
111100 .... 11100 001...	I	..XX			xvnmaddadp	v2.06			722	VSX Vector Negative Multiply-Add Type-A Double-Precision XX3-form
111100 .... 11000 001...	I	..XX			xvnmaddasp	v2.06			726	VSX Vector Negative Multiply-Add Type-A Single-Precision XX3-form
111100 .... 11101 001...	I	..XX			xvnmaddmdp	v2.06			722	VSX Vector Negative Multiply-Add Type-M Double-Precision XX3-form
111100 .... 11001 001...	I	..XX			xvnmaddmsp	v2.06			726	VSX Vector Negative Multiply-Add Type-M Single-Precision XX3-form
111100 .... 11110 001...	I	..XX			xvnmsubadp	v2.06			729	VSX Vector Negative Multiply-Subtract Type-A Double-Precision XX3-form
111100 .... 11010 001...	I	..XX			xvnmsubasp	v2.06			732	VSX Vector Negative Multiply-Subtract Type-A Single-Precision XX3-form
111100 .... 11111 001...	I	..XX			xvnmsubmdp	v2.06			729	VSX Vector Negative Multiply-Subtract Type-M Double-Precision XX3-form
111100 .... 11011 001...	I	..XX			xvnmsubmsp	v2.06			732	VSX Vector Negative Multiply-Subtract Type-M Single-Precision XX3-form
111100 ..... ///// ..... 01100 1001..	I	..XX			xvrdpi	v2.06			808	VSX Vector Round to Double-Precision Integer using round to Nearest Away XX2-form
111100 ..... ///// ..... 01110 1011..	I	..XX			xvrdpic	v2.06			809	VSX Vector Round to Double-Precision Integer Exact using Current rounding mode XX2-form
111100 ..... ///// ..... 01111 1001..	I	..XX			xvrdpim	v2.06			810	VSX Vector Round to Double-Precision Integer using round toward -Infinity XX2-form
111100 ..... ///// ..... 01110 1001..	I	..XX			xvrdpip	v2.06			810	VSX Vector Round to Double-Precision Integer using round toward +Infinity XX2-form
111100 ..... ///// ..... 01101 1001..	I	..XX			xvrdpiz	v2.06			811	VSX Vector Round to Double-Precision Integer using round toward Zero XX2-form
111100 ..... ///// ..... 01101 1010..	I	..XX			xvredp	v2.06			735	VSX Vector Reciprocal Estimate Double-Precision XX2-form
111100 ..... ///// ..... 01001 1010..	I	..XX			xvresp	v2.06			736	VSX Vector Reciprocal Estimate Single-Precision XX2-form
111100 ..... ///// ..... 01000 1001..	I	..XX			xvrspi	v2.06			812	VSX Vector Round to Single-Precision Integer using round to Nearest Away XX2-form
111100 ..... ///// ..... 01010 1011..	I	..XX			xvrspic	v2.06			813	VSX Vector Round to Single-Precision Integer Exact using Current rounding mode XX2-form
111100 ..... ///// ..... 01011 1001..	I	..XX			xvrspim	v2.06			814	VSX Vector Round to Single-Precision Integer using round toward -Infinity XX2-form
111100 ..... ///// ..... 01010 1001..	I	..XX			xvrspip	v2.06			814	VSX Vector Round to Single-Precision Integer using round toward +Infinity XX2-form

Table F.1: Power ISA Instruction Set Sorted by Version (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
111100 ..... ///// ..... 01001 1001..	I	..XX			xvrspiz	v2.06			815	VSX Vector Round to Single-Precision Integer using round toward Zero XX2-form
111100 ..... ///// ..... 01100 1010..	I	..XX			xvrsqrtedp	v2.06			737	VSX Vector Reciprocal Square Root Estimate Double-Precision XX2-form
111100 ..... ///// ..... 01000 1010..	I	..XX			xvrsqrtesp	v2.06			738	VSX Vector Reciprocal Square Root Estimate Single-Precision XX2-form
111100 ..... ///// ..... 01100 1011..	I	..XX			xvsqrtdp	v2.06			704	VSX Vector Square Root Double-Precision XX2-form
111100 ..... ///// ..... 01000 1011..	I	..XX			xvsqrtsp	v2.06			705	VSX Vector Square Root Single-Precision XX2-form
111100 .... 01101 000...	I	..XX			xvsubdp	v2.06			706	VSX Vector Subtract Double-Precision XX3-form
111100 .... 01001 000...	I	..XX			xvsubsp	v2.06			708	VSX Vector Subtract Single-Precision XX3-form
111100 ...// .... 01111 101../	I	..XX			xvtdivdp	v2.06			739	VSX Vector Test for software Divide Double-Precision XX3-form
111100 ...// .... 01011 101../	I	..XX			xvtdivsp	v2.06			740	VSX Vector Test for software Divide Single-Precision XX3-form
111100 ...// ///// 01110 1010./	I	..XX			xvtsqrtdp	v2.06			741	VSX Vector Test for software Square Root Double-Precision XX2-form
111100 ...// ///// 01010 1010./	I	..XX			xvtsqrtsp	v2.06			742	VSX Vector Test for software Square Root Single-Precision XX2-form
111100 .... 10000 010...	I	..XX			xxland	v2.06			907	VSX Vector Logical AND XX3-form
111100 .... 10001 010...	I	..XX			xxlandc	v2.06			907	VSX Vector Logical AND with Complement XX3-form
111100 .... 10100 010...	I	..XX			xxlnor	v2.06			908	VSX Vector Logical NOR XX3-form
111100 .... 10010 010...	I	..XX			xxlor	v2.06			908	VSX Vector Logical OR XX3-form
111100 .... 10011 010...	I	..XX			xxlxor	v2.06			908	VSX Vector Logical XOR XX3-form
111100 .... 00010 010...	I	..XX			xxmrghw	v2.06			918	VSX Vector Merge High Word XX3-form
111100 .... 00110 010...	I	..XX			xxmrglw	v2.06			918	VSX Vector Merge Low Word XX3-form
111100 .... 0..01 010...	I	..XX			xxpermdi	v2.06			922	VSX Vector Permute Doubleword Immediate XX3-form
111100 .... 11....	I	..XX			xxsel	v2.06			909	VSX Vector Select XX4-form
111100 .... 0..00 010...	I	..XX			xxsldwi	v2.06			925	VSX Vector Shift Left Double by Word Immediate XX3-form
111100 ..... ///.. ..... 01010 0100..	I	..XX			xxspltw	v2.06			921	VSX Vector Splat Word XX2-form
011111 .... 01111 11100/	I	XXXX			cmpb	v2.05			101	Compare Bytes X-form
111011 .... 00000 00010.	I	...X	DFP		dadd[.]	v2.05			204	DFP Add X-form
111111 .... 00000 00010.	I	...X	DFP		daddq[.]	v2.05			204	DFP Add Quad X-form
111111 ..... ///// ..... 11001 00010.	I	...X	DFP		dcffixq[.]	v2.05			227	DFP Convert From Fixed Quad X-form
111011 ...// 00100 00010/	I	...X	DFP		dcmpo	v2.05			209	DFP Compare Ordered X-form
111111 ...// 00100 00010/	I	...X	DFP		dcmpoq	v2.05			209	DFP Compare Ordered Quad X-form
111011 ...// 10100 00010/	I	...X	DFP		dcmpu	v2.05			208	DFP Compare Unordered X-form
111111 ...// 10100 00010/	I	...X	DFP		dcmpuq	v2.05			208	DFP Compare Unordered Quad X-form
111011 ..... ///// ..... 01000 00010.	I	...X	DFP		dctdp[.]	v2.05			225	DFP Convert To DFP Long X-form
111011 ..... ///// ..... 01001 00010.	I	...X	DFP		dctfix[.]	v2.05			228	DFP Convert To Fixed X-form
111111 ..... ///// ..... 01001 00010.	I	...X	DFP		dctfixq[.]	v2.05			228	DFP Convert To Fixed Quad X-form
111111 ..... ///// ..... 01000 00010.	I	...X	DFP		dctqpq[.]	v2.05			225	DFP Convert To DFP Extended X-form
111011 ..../// ..... 01010 00010.	I	...X	DFP		ddedpd[.]	v2.05			230	DFP Decode DPD To BCD X-form
111111 ..../// ..... 01010 00010.	I	...X	DFP		ddedpdq[.]	v2.05			230	DFP Decode DPD To BCD Quad X-form
111011 .... 10001 00010.	I	...X	DFP		ddiv[.]	v2.05			207	DFP Divide X-form
111111 .... 10001 00010.	I	...X	DFP		ddivq[.]	v2.05			207	DFP Divide Quad X-form
111011 ..... ./// ..... 11010 00010.	I	...X	DFP		denbcd[.]	v2.05			230	DFP Encode BCD To DPD X-form
111111 ..... ./// ..... 11010 00010.	I	...X	DFP		denbcdq[.]	v2.05			230	DFP Encode BCD To DPD Quad X-form
111011 .... 11011 00010.	I	...X	DFP		diex[.]	v2.05			231	DFP Insert Biased Exponent X-form

Table F.1: Power ISA Instruction Set Sorted by Version (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
111111 .... 11011 00010.	I	...X	DFP		diexq[.]	v2.05			231	DFP Insert Biased Exponent Quad X-form
111011 .... 00001 00010.	I	...X	DFP		dmul[.]	v2.05			206	DFP Multiply X-form
111111 .... 00001 00010.	I	...X	DFP		dmulq[.]	v2.05			206	DFP Multiply Quad X-form
111011 ....000 00011.	I	...X	DFP		dqua[.]	v2.05			215	DFP Quantize Z23-form
111011 ....010 00011.	I	...X	DFP		dquai[.]	v2.05			214	DFP Quantize Immediate Z23-form
111111 ....010 00011.	I	...X	DFP		dquaiq[.]	v2.05			214	DFP Quantize Immediate Quad Z23-form
111111 ....000 00011.	I	...X	DFP		dquaq[.]	v2.05			215	DFP Quantize Quad Z23-form
111111 ..... ///// ..... 11000 00010.	I	...X	DFP		drdpq[.]	v2.05			226	DFP Round To DFP Long X-form
111011 ..... ////. ....111 00011.	I	...X	DFP		drintn[.]	v2.05			222	DFP Round To FP Integer Without Inexact Z23-form
111111 ..... ////. ....111 00011.	I	...X	DFP		drintnq[.]	v2.05			222	DFP Round To FP Integer Without Inexact Quad Z23-form
111011 ..... ////. ....011 00011.	I	...X	DFP		drintx[.]	v2.05			220	DFP Round To FP Integer With Inexact Z23-form
111111 ..... ////. ....011 00011.	I	...X	DFP		drintxq[.]	v2.05			220	DFP Round To FP Integer With Inexact Quad Z23-form
111011 ....001 00011.	I	...X	DFP		drrnd[.]	v2.05			217	DFP Reround Z23-form
111111 ....001 00011.	I	...X	DFP		drrndq[.]	v2.05			217	DFP Reround Quad Z23-form
111011 ..... ///// ..... 11000 00010.	I	...X	DFP		drsp[.]	v2.05			226	DFP Round To DFP Short X-form
111011 .... .0010 00010.	I	...X	DFP		dscli[.]	v2.05			233	DFP Shift Significand Left Immediate Z22-form
111111 .... .0010 00010.	I	...X	DFP		dscliq[.]	v2.05			233	DFP Shift Significand Left Immediate Quad Z22-form
111011 .... .0011 00010.	I	...X	DFP		dscri[.]	v2.05			233	DFP Shift Significand Right Immediate Z22-form
111111 .... .0011 00010.	I	...X	DFP		dscriq[.]	v2.05			233	DFP Shift Significand Right Immediate Quad Z22-form
111011 .... 10000 00010.	I	...X	DFP		dsub[.]	v2.05			205	DFP Subtract X-form
111111 .... 10000 00010.	I	...X	DFP		dsubq[.]	v2.05			205	DFP Subtract Quad X-form
111011 ...// .... .0110 00010/	I	...X	DFP		dtstdc	v2.05			210	DFP Test Data Class Z22-form
111111 ...// .... .0110 00010/	I	...X	DFP		dtstdcq	v2.05			210	DFP Test Data Class Quad Z22-form
111011 ...// .... .0111 00010/	I	...X	DFP		dtstdg	v2.05			210	DFP Test Data Group Z22-form
111111 ...// .... .0111 00010/	I	...X	DFP		dtstdgq	v2.05			210	DFP Test Data Group Quad Z22-form
111011 ...// 00101 00010/	I	...X	DFP		dtstex	v2.05			211	DFP Test Exponent X-form
111111 ...// 00101 00010/	I	...X	DFP		dtstexq	v2.05			211	DFP Test Exponent Quad X-form
111011 ...// 10101 00010/	I	...X	DFP		dtstsf	v2.05			212	DFP Test Significance X-form
111111 ...// 10101 00010/	I	...X	DFP		dtstsfq	v2.05			212	DFP Test Significance Quad X-form
111011 ..... ///// ..... 01011 00010.	I	...X	DFP		dxex[.]	v2.05			231	DFP Extract Biased Exponent X-form
111111 ..... ///// ..... 01011 00010.	I	...X	DFP		dxexq[.]	v2.05			231	DFP Extract Biased Exponent Quad X-form
111111 .... 00000 01000.	I	.XXX			fcpsgn[.]	v2.05			160	Floating Copy Sign X-form
011111 .... 11010 10101/	III	..XX			lbzcix	v2.05	HV		1097	Load Byte and Zero Caching Inhibited Indexed X-form
011111 .... 11011 10101/	III	..XX			ldcix	v2.05	HV		1097	Load Doubleword Caching Inhibited Indexed X-form
111001 00	I	...X			lfdp	v2.05			158	Load Floating-Point Double Pair DS-form
011111 .... 11000 10111/	I	...X			lfdpx	v2.05			158	Load Floating-Point Double Pair Indexed X-form
011111 .... 11010 10111/	I	.XXX			lfiwax	v2.05			153	Load Floating-Point as Integer Word Algebraic Indexed X-form
011111 .... 11001 10101/	III	..XX			lhzcix	v2.05	HV		1097	Load Halfword and Zero Caching Inhibited Indexed X-form
011111 .... 11000 10101/	III	..XX			lwzcix	v2.05	HV		1097	Load Word and Zero Caching Inhibited Indexed X-form
011111 .... ///// 00101 11010/	I	..XX			prtyd	v2.05			103	Parity Doubleword X-form
011111 .... ///// 00100 11010/	I	XXXX			prtyw	v2.05			102	Parity Word X-form
011111 ..... ///// 11110 100111	III	...X			slbfee.	v2.05	P	SR	1170	SLB Find Entry ESID X-form
011111 .... 11110 10101/	III	..XX			stbcix	v2.05	HV		1098	Store Byte Caching Inhibited Indexed X-form
011111 .... 11111 10101/	III	..XX			stdcix	v2.05	HV		1098	Store Doubleword Caching Inhibited Indexed X-form
111101 00	I	...X			stfdp	v2.05			159	Store Floating-Point Double Pair DS-form
011111 .... 11100 10111/	I	...X			stfdpx	v2.05			159	Store Floating-Point Double Pair Indexed X-form

Table F.1: Power ISA Instruction Set Sorted by Version (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
011111 .... 11101 10101/	III	..XX			sthcix	v2.05	HV		1098	Store Halfword Caching Inhibited Indexed X-form
011111 .... 11100 10101/	III	..XX			stwcix	v2.05	HV		1098	Store Word Caching Inhibited Indexed X-form
011111 .... 01111/	I	XXXX			isel	v2.03			96	Integer Select A-form
111000	I	..XX			lq	v2.03			63	Load Quadword DQ-form
011111 .... 00000 00111/	I	..XX			lvebx	v2.03			260	Load Vector Element Byte Indexed X-form
011111 .... 00001 00111/	I	..XX			lvehx	v2.03			260	Load Vector Element Halfword Indexed X-form
011111 .... 00010 00111/	I	..XX			lvewx	v2.03			261	Load Vector Element Word Indexed X-form
011111 .... 00000 00110/	I	..XX			lvsl	v2.03			267	Load Vector for Shift Left Indexed X-form
011111 .... 00001 00110/	I	..XX			lvsr	v2.03			267	Load Vector for Shift Right Indexed X-form
011111 .... 00011 00111/	I	..XX			lvx	v2.03			262	Load Vector Indexed X-form
011111 .... 01011 00111/	I	..XX			lvxl	v2.03			262	Load Vector Indexed Last X-form
000100 ..... ///// ///// 11000 000100	I	..XX			mfvscr	v2.03			483	Move From Vector Status and Control Register VX-form
000100 ///// ///// 11001 000100	I	..XX			mtvscr	v2.03			483	Move To Vector Status and Control Register VX-form
111110 10	I	..XX			stq	v2.03			64	Store Quadword DS-form
011111 .... 00100 00111/	I	..XX			stvebx	v2.03			263	Store Vector Element Byte Indexed X-form
011111 .... 00101 00111/	I	..XX			stvehx	v2.03			264	Store Vector Element Halfword Indexed X-form
011111 .... 00110 00111/	I	..XX			stvewx	v2.03			264	Store Vector Element Word Indexed X-form
011111 .... 00111 00111/	I	..XX			stvx	v2.03			265	Store Vector Indexed X-form
011111 .... 01111 00111/	I	..XX			stvxl	v2.03			265	Store Vector Indexed Last X-form
011111 ..... /.... ..... 01000 10010/	III	..XX			tlbiel	v2.03	P	64	1181	TLB Invalidate Entry Local X-form
000100 .... 00110 000000	I	..XX			vaddcuw	v2.03			313	Vector Add & write Carry-out Unsigned Word VX-form
000100 .... 00000 001010	I	..XX			vaddfp	v2.03			411	Vector Add Floating-Point VX-form
000100 .... 01100 000000	I	..XX			vaddsbs	v2.03			313	Vector Add Signed Byte Saturate VX-form
000100 .... 01101 000000	I	..XX			vaddshs	v2.03			314	Vector Add Signed Halfword Saturate VX-form
000100 .... 01110 000000	I	..XX			vaddsws	v2.03			314	Vector Add Signed Word Saturate VX-form
000100 .... 00000 000000	I	..XX			vaddubm	v2.03			315	Vector Add Unsigned Byte Modulo VX-form
000100 .... 01000 000000	I	..XX			vaddubs	v2.03			317	Vector Add Unsigned Byte Saturate VX-form
000100 .... 00001 000000	I	..XX			vadduhm	v2.03			315	Vector Add Unsigned Halfword Modulo VX-form
000100 .... 01001 000000	I	..XX			vadduhs	v2.03			317	Vector Add Unsigned Halfword Saturate VX-form
000100 .... 00010 000000	I	..XX			vadduwm	v2.03			316	Vector Add Unsigned Word Modulo VX-form
000100 .... 01010 000000	I	..XX			vadduws	v2.03			318	Vector Add Unsigned Word Saturate VX-form
000100 .... 10000 000100	I	..XX			vand	v2.03			392	Vector Logical AND VX-form
000100 .... 10001 000100	I	..XX			vandc	v2.03			392	Vector Logical AND with Complement VX-form
000100 .... 10100 000010	I	..XX			vavgsb	v2.03			365	Vector Average Signed Byte VX-form
000100 .... 10101 000010	I	..XX			vavgsh	v2.03			366	Vector Average Signed Halfword VX-form
000100 .... 10110 000010	I	..XX			vavgsw	v2.03			367	Vector Average Signed Word VX-form
000100 .... 10000 000010	I	..XX			vavgub	v2.03			365	Vector Average Unsigned Byte VX-form
000100 .... 10001 000010	I	..XX			vavguh	v2.03			366	Vector Average Unsigned Halfword VX-form
000100 .... 10010 000010	I	..XX			vavguw	v2.03			367	Vector Average Unsigned Word VX-form
000100 .... 01101 001010	I	..XX			vcfsx	v2.03			415	Vector Convert with round to nearest From Signed Word to floating-point format VX-form
000100 .... 01100 001010	I	..XX			vcfux	v2.03			415	Vector Convert with round to nearest From Unsigned Word to floating-point format VX-form
000100 .... .1111 000110	I	..XX			vcmpbfp[.]	v2.03			418	Vector Compare Bounds Floating-Point VC-form
000100 .... .0011 000110	I	..XX			vcmpeqfp[.]	v2.03			419	Vector Compare Equal Floating-Point VC-form
000100 .... .0000 000110	I	..XX			vcmpequb[.]	v2.03			378	Vector Compare Equal Unsigned Byte VC-form
000100 .... .0001 000110	I	..XX			vcmpequh[.]	v2.03			379	Vector Compare Equal Unsigned Halfword VC-form
000100 .... .0010 000110	I	..XX			vcmpequw[.]	v2.03			380	Vector Compare Equal Unsigned Word VC-form

Table F.1: Power ISA Instruction Set Sorted by Version (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
000100 .... .0111 000110	I	..XX			vcmpgefp[.]	v2.03			419	Vector Compare Greater Than or Equal Floating-Point VC-form
000100 .... .1011 000110	I	..XX			vcmpgtfp[.]	v2.03			420	Vector Compare Greater Than Floating-Point VC-form
000100 .... .1100 000110	I	..XX			vcmpgtsb[.]	v2.03			383	Vector Compare Greater Than Signed Byte VC-form
000100 .... .1101 000110	I	..XX			vcmpgtsh[.]	v2.03			384	Vector Compare Greater Than Signed Halfword VC-form
000100 .... .1110 000110	I	..XX			vcmpgtsw[.]	v2.03			385	Vector Compare Greater Than Signed Word VC-form
000100 .... .1000 000110	I	..XX			vcmpgtub[.]	v2.03			383	Vector Compare Greater Than Unsigned Byte VC-form
000100 .... .1001 000110	I	..XX			vcmpgtuh[.]	v2.03			384	Vector Compare Greater Than Unsigned Halfword VC-form
000100 .... .1010 000110	I	..XX			vcmpgtuw[.]	v2.03			385	Vector Compare Greater Than Unsigned Word VC-form
000100 .... 01111 001010	I	..XX			vctsxs	v2.03			414	Vector Convert with round to zero from floating-point To Signed Word format Saturate VX-form
000100 .... 01110 001010	I	..XX			vctuxs	v2.03			414	Vector Convert with round to zero from floating-point To Unsigned Word format Saturate VX-form
000100 ..... ///// 00110 001010	I	..XX			vexptefp	v2.03			421	Vector 2 Raised to the Exponent Estimate Floating-Point VX-form
000100 ..... ///// 00111 001010	I	..XX			vlogefp	v2.03			422	Vector Log Base 2 Estimate Floating-Point VX-form
000100 101110	I	..XX			vmaddfp	v2.03			412	Vector Multiply-Add Floating-Point VA-form
000100 .... 10000 001010	I	..XX			vmaxfp	v2.03			413	Vector Maximum Floating-Point VX-form
000100 .... 00100 000010	I	..XX			vmaxsb	v2.03			370	Vector Maximum Signed Byte VX-form
000100 .... 00101 000010	I	..XX			vmaxsh	v2.03			371	Vector Maximum Signed Halfword VX-form
000100 .... 00110 000010	I	..XX			vmaxsw	v2.03			372	Vector Maximum Signed Word VX-form
000100 .... 00000 000010	I	..XX			vmaxub	v2.03			370	Vector Maximum Unsigned Byte VX-form
000100 .... 00001 000010	I	..XX			vmaxuh	v2.03			371	Vector Maximum Unsigned Halfword VX-form
000100 .... 00010 000010	I	..XX			vmaxuw	v2.03			372	Vector Maximum Unsigned Word VX-form
000100 100000	I	..XX			vmhaddshs	v2.03			341	Vector Multiply-High-Add Signed Halfword Saturate VA-form
000100 100001	I	..XX			vmhraddshs	v2.03			341	Vector Multiply-High-Round-Add Signed Halfword Saturate VA-form
000100 .... 10001 001010	I	..XX			vminfp	v2.03			413	Vector Minimum Floating-Point VX-form
000100 .... 01100 000010	I	..XX			vminsb	v2.03			374	Vector Minimum Signed Byte VX-form
000100 .... 01101 000010	I	..XX			vminsh	v2.03			375	Vector Minimum Signed Halfword VX-form
000100 .... 01110 000010	I	..XX			vminsw	v2.03			376	Vector Minimum Signed Word VX-form
000100 .... 01000 000010	I	..XX			vminub	v2.03			374	Vector Minimum Unsigned Byte VX-form
000100 .... 01001 000010	I	..XX			vminuh	v2.03			375	Vector Minimum Unsigned Halfword VX-form
000100 .... 01010 000010	I	..XX			vminuw	v2.03			376	Vector Minimum Unsigned Word VX-form
000100 100010	I	..XX			vmladduhm	v2.03			342	Vector Multiply-Low-Add Unsigned Halfword Modulo VA-form
000100 .... 00000 001100	I	..XX			vmrghb	v2.03			279	Vector Merge High Byte VX-form
000100 .... 00001 001100	I	..XX			vmrghh	v2.03			280	Vector Merge High Halfword VX-form
000100 .... 00010 001100	I	..XX			vmrghw	v2.03			281	Vector Merge High Word VX-form
000100 .... 00100 001100	I	..XX			vmrglb	v2.03			279	Vector Merge Low Byte VX-form
000100 .... 00101 001100	I	..XX			vmrglh	v2.03			280	Vector Merge Low Halfword VX-form
000100 .... 00110 001100	I	..XX			vmrglw	v2.03			281	Vector Merge Low Word VX-form
000100 100101	I	..XX			vmsummbm	v2.03			343	Vector Multiply-Sum Mixed Byte Modulo VA-form
000100 101000	I	..XX			vmsumshm	v2.03			343	Vector Multiply-Sum Signed Halfword Modulo VA-form
000100 101001	I	..XX			vmsumshs	v2.03			344	Vector Multiply-Sum Signed Halfword Saturate VA-form
000100 100100	I	..XX			vmsumubm	v2.03			342	Vector Multiply-Sum Unsigned Byte Modulo VA-form
000100 100110	I	..XX			vmsumuhm	v2.03			344	Vector Multiply-Sum Unsigned Halfword Modulo VA-form

Table F.1: Power ISA Instruction Set Sorted by Version (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
000100 100111	I	..XX			vmsumuhs	v2.03			345	Vector Multiply-Sum Unsigned Halfword Saturate VA-form
000100 .... 01100 001000	I	..XX			vmulesb	v2.03			329	Vector Multiply Even Signed Byte VX-form
000100 .... 01101 001000	I	..XX			vmulesh	v2.03			331	Vector Multiply Even Signed Halfword VX-form
000100 .... 01000 001000	I	..XX			vmuleub	v2.03			330	Vector Multiply Even Unsigned Byte VX-form
000100 .... 01001 001000	I	..XX			vmuleuh	v2.03			332	Vector Multiply Even Unsigned Halfword VX-form
000100 .... 00100 001000	I	..XX			vmulosb	v2.03			329	Vector Multiply Odd Signed Byte VX-form
000100 .... 00101 001000	I	..XX			vmulosh	v2.03			331	Vector Multiply Odd Signed Halfword VX-form
000100 .... 00000 001000	I	..XX			vmuloub	v2.03			330	Vector Multiply Odd Unsigned Byte VX-form
000100 .... 00001 001000	I	..XX			vmulouh	v2.03			332	Vector Multiply Odd Unsigned Halfword VX-form
000100 101111	I	..XX			vnmsubfp	v2.03			412	Vector Negative Multiply-Subtract Floating-Point VA-form
000100 .... 10100 000100	I	..XX			vnor	v2.03			394	Vector Logical NOR VX-form
000100 .... 10010 000100	I	..XX			vor	v2.03			393	Vector Logical OR VX-form
000100 101011	I	..XX			vperm	v2.03			286	Vector Permute VA-form
000100 .... 01100 001110	I	..XX			vpkpx	v2.03			268	Vector Pack Pixel VX-form
000100 .... 00110 001110	I	..XX			vpkshss	v2.03			269	Vector Pack Signed Halfword Signed Saturate VX-form
000100 .... 00100 001110	I	..XX			vpkshus	v2.03			269	Vector Pack Signed Halfword Unsigned Saturate VX-form
000100 .... 00111 001110	I	..XX			vpkswss	v2.03			270	Vector Pack Signed Word Signed Saturate VX-form
000100 .... 00101 001110	I	..XX			vpkswus	v2.03			270	Vector Pack Signed Word Unsigned Saturate VX-form
000100 .... 00000 001110	I	..XX			vpkuhum	v2.03			272	Vector Pack Unsigned Halfword Unsigned Modulo VX-form
000100 .... 00010 001110	I	..XX			vpkuhus	v2.03			272	Vector Pack Unsigned Halfword Unsigned Saturate VX-form
000100 .... 00001 001110	I	..XX			vpkuwum	v2.03			273	Vector Pack Unsigned Word Unsigned Modulo VX-form
000100 .... 00011 001110	I	..XX			vpkuwus	v2.03			273	Vector Pack Unsigned Word Unsigned Saturate VX-form
000100 ..... ///// 00100 001010	I	..XX			vrefp	v2.03			423	Vector Reciprocal Estimate Floating-Point VX-form
000100 ..... ///// 01011 001010	I	..XX			vrfim	v2.03			416	Vector Round to Floating-Point Integer toward -Infinity VX-form
000100 ..... ///// 01000 001010	I	..XX			vrfin	v2.03			416	Vector Round to Floating-Point Integer Nearest VX-form
000100 ..... ///// 01010 001010	I	..XX			vrfip	v2.03			417	Vector Round to Floating-Point Integer toward +Infinity VX-form
000100 ..... ///// 01001 001010	I	..XX			vrfiz	v2.03			417	Vector Round to Floating-Point Integer toward Zero VX-form
000100 .... 00000 000100	I	..XX			vrlb	v2.03			395	Vector Rotate Left Byte VX-form
000100 .... 00001 000100	I	..XX			vrlh	v2.03			395	Vector Rotate Left Halfword VX-form
000100 .... 00010 000100	I	..XX			vrlw	v2.03			396	Vector Rotate Left Word VX-form
000100 ..... ///// 00101 001010	I	..XX			vrsqrtefp	v2.03			423	Vector Reciprocal Square Root Estimate Floating-Point VX-form
000100 101010	I	..XX			vsel	v2.03			287	Vector Select VA-form
000100 .... 00111 000100	I	..XX			vsl	v2.03			290	Vector Shift Left VX-form
000100 .... 00100 000100	I	..XX			vslb	v2.03			402	Vector Shift Left Byte VX-form
000100 .... /.... 101100	I	..XX			vsldoi	v2.03			289	Vector Shift Left Double by Octet Immediate VA-form
000100 .... 00101 000100	I	..XX			vslh	v2.03			402	Vector Shift Left Halfword VX-form
000100 .... 10000 001100	I	..XX			vslo	v2.03			291	Vector Shift Left by Octet VX-form
000100 .... 00110 000100	I	..XX			vslw	v2.03			403	Vector Shift Left Word VX-form
000100 ..... /.... ..... 01000 001100	I	..XX			vspltb	v2.03			283	Vector Splat Byte VX-form
000100 ..... //... ..... 01001 001100	I	..XX			vsplth	v2.03			283	Vector Splat Halfword VX-form
000100 .... ///// 01100 001100	I	..XX			vspltisb	v2.03			285	Vector Splat Immediate Signed Byte VX-form
000100 .... ///// 01101 001100	I	..XX			vspltish	v2.03			285	Vector Splat Immediate Signed Halfword VX-form

Table F.1: Power ISA Instruction Set Sorted by Version (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
000100 .... ///// 01110 001100	I	..XX			vspltisw	v2.03			285	Vector Splat Immediate Signed Word VX-form
000100 ..... ///.. ..... 01010 001100	I	..XX			vspltw	v2.03			284	Vector Splat Word VX-form
000100 .... 01011 000100	I	..XX			vsr	v2.03			290	Vector Shift Right VX-form
000100 .... 01100 000100	I	..XX			vsrab	v2.03			408	Vector Shift Right Algebraic Byte VX-form
000100 .... 01101 000100	I	..XX			vsrah	v2.03			408	Vector Shift Right Algebraic Halfword VX-form
000100 .... 01110 000100	I	..XX			vsraw	v2.03			409	Vector Shift Right Algebraic Word VX-form
000100 .... 01000 000100	I	..XX			vsrb	v2.03			405	Vector Shift Right Byte VX-form
000100 .... 01001 000100	I	..XX			vsrh	v2.03			405	Vector Shift Right Halfword VX-form
000100 .... 10001 001100	I	..XX			vsro	v2.03			291	Vector Shift Right by Octet VX-form
000100 .... 01010 000100	I	..XX			vsrw	v2.03			406	Vector Shift Right Word VX-form
000100 .... 10110 000000	I	..XX			vsubcuw	v2.03			321	Vector Subtract & Write Carry-out Unsigned Word VX-form
000100 .... 00001 001010	I	..XX			vsubfp	v2.03			411	Vector Subtract Floating-Point VX-form
000100 .... 11100 000000	I	..XX			vsubsbs	v2.03			321	Vector Subtract Signed Byte Saturate VX-form
000100 .... 11101 000000	I	..XX			vsubshs	v2.03			322	Vector Subtract Signed Halfword Saturate VX-form
000100 .... 11110 000000	I	..XX			vsubsws	v2.03			322	Vector Subtract Signed Word Saturate VX-form
000100 .... 10000 000000	I	..XX			vsububm	v2.03			323	Vector Subtract Unsigned Byte Modulo VX-form
000100 .... 11000 000000	I	..XX			vsububs	v2.03			325	Vector Subtract Unsigned Byte Saturate VX-form
000100 .... 10001 000000	I	..XX			vsubuhm	v2.03			323	Vector Subtract Unsigned Halfword Modulo VX-form
000100 .... 11001 000000	I	..XX			vsubuhs	v2.03			325	Vector Subtract Unsigned Halfword Saturate VX-form
000100 .... 10010 000000	I	..XX			vsubuwm	v2.03			324	Vector Subtract Unsigned Word Modulo VX-form
000100 .... 11010 000000	I	..XX			vsubuws	v2.03			326	Vector Subtract Unsigned Word Saturate VX-form
000100 .... 11010 001000	I	..XX			vsum2sws	v2.03			358	Vector Sum across Half Signed Word Saturate VX-form
000100 .... 11100 001000	I	..XX			vsum4sbs	v2.03			359	Vector Sum across Quarter Signed Byte Saturate VX-form
000100 .... 11001 001000	I	..XX			vsum4shs	v2.03			359	Vector Sum across Quarter Signed Halfword Saturate VX-form
000100 .... 11000 001000	I	..XX			vsum4ubs	v2.03			360	Vector Sum across Quarter Unsigned Byte Saturate VX-form
000100 .... 11110 001000	I	..XX			vsumsws	v2.03			357	Vector Sum across Signed Word Saturate VX-form
000100 ..... ///// 01101 001110	I	..XX			vupkhpx	v2.03			278	Vector Unpack High Pixel VX-form
000100 ..... ///// 01000 001110	I	..XX			vupkhsb	v2.03			275	Vector Unpack High Signed Byte VX-form
000100 ..... ///// 01001 001110	I	..XX			vupkhsh	v2.03			276	Vector Unpack High Signed Halfword VX-form
000100 ..... ///// 01111 001110	I	..XX			vupklpx	v2.03			278	Vector Unpack Low Pixel VX-form
000100 ..... ///// 01010 001110	I	..XX			vupklsb	v2.03			275	Vector Unpack Low Signed Byte VX-form
000100 ..... ///// 01011 001110	I	..XX			vupklsh	v2.03			276	Vector Unpack Low Signed Halfword VX-form
000100 .... 10011 000100	I	..XX			vxor	v2.03			394	Vector Logical XOR VX-form
011111 ///.. ///.. ///// 00000 11110/	II	..XX			wait	v2.03			1029	Wait X-form
111111 ..... ///// ..... ///// 11000.	I	.XXX			fre[.]	v2.02			165	Floating Reciprocal Estimate A-form
111111 ..... ///// ..... 01111 01000.	I	.XXX			frim[.]	v2.02			178	Floating Round to Integer Minus X-form
111111 ..... ///// ..... 01100 01000.	I	.XXX			frin[.]	v2.02			177	Floating Round to Integer Nearest X-form
111111 ..... ///// ..... 01110 01000.	I	.XXX			frip[.]	v2.02			178	Floating Round to Integer Plus X-form
111111 ..... ///// ..... 01101 01000.	I	.XXX			friz[.]	v2.02			177	Floating Round to Integer Toward Zero X-form
111011 ..... ///// ..... ///// 11010.	I	.XXX			frsqrtes[.]	v2.02			166	Floating Reciprocal Square Root Estimate Single A-form
010011 ///// ///// ///// 01000 10010/	III	..XX			hrfid	v2.02	HV		1086	Hypervisor Return From Interrupt Doubleword XL-form
011111 .... ///// 00011 11010/	I	XXXX			popcntb	v2.02			102	Population Count Bytes X-form
011111 ..... 1.... ..../ 00000 10011/	I	XXXX			mfocrf	v2.01			130	Move From One Condition Register Field XFX-form
011111 ..... 1.... ..../ 00100 10000/	I	XXXX			mtocrf	v2.01			129	Move To One Condition Register Field XFX-form
011111 ..... ///// 11100 10011/	III	...X			slbmfee	v2.00	P		1169	SLB Move From Entry ESID X-form

Table F.1: Power ISA Instruction Set Sorted by Version (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
011111 ..... ///// 11010 10011/	III	...X			slbmfev	v2.00	P		1169	SLB Move From Entry VSID X-form
011111 ..... ///// 01100 10010/	III	...X			slbmte	v2.00	P		1168	SLB Move To Entry X-form
011111 .... ///// 00001 11010.	I	..XX			cntlzd[.]	PPC		SR	104	Count Leading Zeros Doubleword X-form
011111 //... .... 00010 10110/	II	..XX			dcbf	PPC			1003	Data Cache Block Flush X-form
011111 ///// 00001 10110/	II	..XX			dcbst	PPC			1002	Data Cache Block Store X-form
011111 .... 01000 10110/	II	..XX			dcbt	PPC			999	Data Cache Block Touch X-form
011111 .... 00111 10110/	II	..XX			dcbtst	PPC			1000	Data Cache Block Touch for Store X-form
011111 .... 01111 01001.	I	..XX			divd[.]	PPC		SR	87	Divide Doubleword XO-form
011111 .... 11111 01001.	I	..XX			divdo[.]	PPC		SR	87	Divide Doubleword & record OV XO-form
011111 .... 01110 01001.	I	..XX			divdu[.]	PPC		SR	87	Divide Doubleword Unsigned XO-form
011111 .... 11110 01001.	I	..XX			divduo[.]	PPC		SR	87	Divide Doubleword Unsigned & record OV XO-form
011111 .... 01111 01011.	I	XXXX			divw[.]	PPC		SR	80	Divide Word XO-form
011111 .... 11111 01011.	I	..XX			divwo[.]	PPC		SR	80	Divide Word & record OV XO-form
011111 .... 01110 01011.	I	XXXX			divwu[.]	PPC		SR	80	Divide Word Unsigned XO-form
011111 .... 11110 01011.	I	..XX			divwuo[.]	PPC		SR	80	Divide Word Unsigned & record OV XO-form
011111 ///// ///// ///// 11010 10110/	II	XXXX			eieio	PPC			1028	Enforce In-order Execution of I/O X-form
011111 .... ///// 11101 11010.	I	XXXX			extsb[.]	PPC		SR	101	Extend Sign Byte X-form
011111 .... ///// 11110 11010.	I	..XX			extsw[.]	PPC		SR	103	Extend Sign Word X-form
111011 .... ///// 10101.	I	.XXX			fadds[.]	PPC			162	Floating Add Single A-form
111111 ..... ///// ..... 11010 01110.	I	.XXX			fcfid[.]	PPC			175	Floating Convert with round Signed Doubleword to Double-Precision format X-form
111111 ..... ///// ..... 11001 01110.	I	.XXX			fctid[.]	PPC			171	Floating Convert with round Double-Precision To Signed Doubleword format X-form
111111 ..... ///// ..... 11001 01111.	I	.XXX			fctidz[.]	PPC			171	Floating Convert with truncate Double-Precision To Signed Doubleword format X-form
111011 .... ///// 10010.	I	.XXX			fdivs[.]	PPC			163	Floating Divide Single A-form
111011 .... 11101.	I	.XXX			fmadds[.]	PPC			168	Floating Multiply-Add Single A-form
111011 .... 11100.	I	.XXX			fmsubs[.]	PPC			168	Floating Multiply-Subtract Single A-form
111011 .... ///// ..... 11001.	I	.XXX			fmuls[.]	PPC			163	Floating Multiply Single A-form
111011 .... 11111.	I	.XXX			fnmadds[.]	PPC			169	Floating Negative Multiply-Add Single A-form
111011 .... 11110.	I	.XXX			fnmsubs[.]	PPC			169	Floating Negative Multiply-Subtract Single A-form
111011 ..... ///// ..... ///// 11000.	I	.XXX			fres[.]	PPC			165	Floating Reciprocal Estimate Single A-form
111111 ..... ///// ..... ///// 11010.	I	.XXX			frsqrte[.]	PPC			166	Floating Reciprocal Square Root Estimate A-form
111111 .... 10111.	I	.XXX			fsel[.]	PPC			180	Floating Select A-form
111011 ..... ///// ..... ///// 10110.	I	.XXX			fsqrts[.]	PPC			164	Floating Square Root Single A-form
111011 .... ///// 10100.	I	.XXX			fsubs[.]	PPC			162	Floating Subtract Single A-form
011111 ///// 11110 10110/	II	..XX			icbi	PPC			991	Instruction Cache Block Invalidate X-form
111010 00	I	..XX			ld	PPC			55	Load Doubleword DS-form
011111 .... 00010 10100.	II	..XX			ldarx	PPC			1021	Load Doubleword And Reserve Indexed X-form
111010 01	I	..XX			ldu	PPC			55	Load Doubleword with Update DS-form
011111 .... 00001 10101/	I	..XX			ldux	PPC			55	Load Doubleword with Update Indexed X-form
011111 .... 00000 10101/	I	..XX			ldx	PPC			55	Load Doubleword Indexed X-form
111010 10	I	..XX			lwa	PPC			54	Load Word Algebraic DS-form
011111 .... 00000 10100.	II	XXXX			lwarx	PPC			1017	Load Word And Reserve Indexed X-form
011111 .... 01011 10101/	I	..XX			lwaux	PPC			54	Load Word Algebraic with Update Indexed X-form
011111 .... 01010 10101/	I	..XX			lwax	PPC			54	Load Word Algebraic Indexed X-form
011111 .... 01011 10011/	II	XXXX			mftb	PPC			1032	Move From Time Base XFX-form
011111 ..... ////. ///// 00101 10010/	III	..XX			mtmsrd	PPC	P		1109	Move To Machine State Register Doubleword X-form

Table F.1: Power ISA Instruction Set Sorted by Version (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
011111 .... /0010 01001.	I	..XX			mulhd[.]	PPC		SR	85	Multiply High Doubleword XO-form
011111 .... /0000 01001.	I	..XX			mulhdu[.]	PPC		SR	85	Multiply High Doubleword Unsigned XO-form
011111 .... /0010 01011.	I	XXXX			mulhw[.]	PPC		SR	79	Multiply High Word XO-form
011111 .... /0000 01011.	I	XXXX			mulhwu[.]	PPC		SR	79	Multiply High Word Unsigned XO-form
011111 .... 00111 01001.	I	..XX			mulld[.]	PPC		SR	85	Multiply Low Doubleword XO-form
011111 .... 10111 01001.	I	..XX			mulldo[.]	PPC		SR	85	Multiply Low Doubleword & record OV XO-form
010011 ///// ///// ///// 00000 10010/	III	XXXX			rfid	PPC	P		1086	Return From Interrupt Doubleword XL-form
011110 .... .1000.	I	..XX			rldcl[.]	PPC		SR	111	Rotate Left Doubleword then Clear Left MDS-form
011110 .... .1001.	I	..XX			rldcr[.]	PPC		SR	112	Rotate Left Doubleword then Clear Right MDS-form
011110 .... .010..	I	..XX			rldic[.]	PPC		SR	111	Rotate Left Doubleword Immediate then Clear MD-form
011110 .... .000..	I	..XX			rldicl[.]	PPC		SR	110	Rotate Left Doubleword Immediate then Clear Left MD-form
011110 .... .001..	I	..XX			rldicr[.]	PPC		SR	110	Rotate Left Doubleword Immediate then Clear Right MD-form
011110 .... .011..	I	..XX			rldimi[.]	PPC		SR	112	Rotate Left Doubleword Immediate then Mask Insert MD-form
010001 ///// ///// ////. ..... .///1/	I	XXXX			sc	PPC			45	System Call SC-form
011111 //... ///// ///// 01111 10010/	III	..XX			slbia	PPC	P		1164	SLB Invalidate All X-form
011111 ///// ///// 01101 10010/	III	...X			slbie	PPC	P		1160	SLB Invalidate Entry X-form
011111 .... 00000 11011.	I	..XX			sld[.]	PPC		SR	115	Shift Left Doubleword X-form
011111 .... 11000 11010.	I	..XX			srad[.]	PPC		SR	115	Shift Right Algebraic Doubleword X-form
011111 .... 11001 1101..	I	..XX			sradi[.]	PPC		SR	115	Shift Right Algebraic Doubleword Immediate XS-form
011111 .... 10000 11011.	I	..XX			srd[.]	PPC		SR	115	Shift Right Doubleword X-form
111110 00	I	..XX			std	PPC			60	Store Doubleword DS-form
011111 .... 00110 101101	II	..XX			stdcx.	PPC			1022	Store Doubleword Conditional Indexed X-form
111110 01	I	..XX			stdu	PPC			61	Store Doubleword with Update DS-form
011111 .... 00101 10101/	I	..XX			stdux	PPC			61	Store Doubleword with Update Indexed X-form
011111 .... 00100 10101/	I	..XX			stdx	PPC			60	Store Doubleword Indexed X-form
011111 .... 11110 10111/	I	.XXX			stfiwx	PPC			157	Store Floating-Point as Integer Word Indexed X-form
011111 .... 00100 101101	II	XXXX			stwcx.	PPC			1020	Store Word Conditional Indexed X-form
011111 .... 00001 01000.	I	XXXX			subf[.]	PPC		SR	75	Subtract From XO-form
011111 .... 10001 01000.	I	..XX			subfo[.]	PPC		SR	75	Subtract From & record OV XO-form
011111 .... 00010 00100/	I	..XX			td	PPC			95	Trap Doubleword X-form
000010	I	..XX			tdi	PPC			95	Trap Doubleword Immediate D-form
011111 ///// ///// ///// 10001 10110/	III	...X			tlbsync	PPC	HV/P		1186	TLB Synchronize X-form
111111 ..... ///// ..... 00000 01110.	I	.XXX			fctiw[.]	P2			173	Floating Convert with round Double-Precision To Signed Word format X-form
111111 ..... ///// ..... 00000 01111.	I	.XXX			fctiwz[.]	P2			173	Floating Convert with truncate Double-Precision To Signed Word format X-form
111111 ..... ///// ..... ///// 10110.	I	.XXX			fsqrt[.]	P2			164	Floating Square Root A-form
011111 .... 01000 01010.	I	XXXX			add[.]	P1		SR	74	Add XO-form
011111 .... 00000 01010.	I	XXXX			addc[.]	P1		SR	76	Add Carrying XO-form
011111 .... 10000 01010.	I	..XX			addco[.]	P1		SR	76	Add Carrying & record OV XO-form
011111 .... 00100 01010.	I	XXXX			adde[.]	P1		SR	76	Add Extended XO-form
011111 .... 10100 01010.	I	..XX			addeo[.]	P1		SR	76	Add Extended & record OV XO-form
001110	I	XXXX			addi	P1			73	Add Immediate D-form
001100	I	XXXX			addic	P1		SR	75	Add Immediate Carrying D-form
001101	I	XXXX			addic.	P1		SR	75	Add Immediate Carrying and Record D-form

Table F.1: Power ISA Instruction Set Sorted by Version (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
001111	I	XXXX			addis	P1			74	Add Immediate Shifted D-form
011111 .... ///// 00111 01010.	I	XXXX			addme[.]	P1		SR	77	Add to Minus One Extended XO-form
011111 .... ///// 10111 01010.	I	..XX			addmeo[.]	P1		SR	77	Add to Minus One Extended & record OV XO-form
011111 .... 11000 01010.	I	..XX			addo[.]	P1		SR	74	Add & record OV XO-form
011111 .... ///// 00110 01010.	I	XXXX			addze[.]	P1		SR	77	Add to Zero Extended XO-form
011111 .... ///// 10110 01010.	I	..XX			addzeo[.]	P1		SR	77	Add to Zero Extended & record OV XO-form
011111 .... 00000 11100.	I	XXXX			and[.]	P1		SR	99	AND X-form
011111 .... 00001 11100.	I	XXXX			andc[.]	P1		SR	100	AND with Complement X-form
011100	I	XXXX			andi.	P1		SR	97	AND Immediate D-form
011101	I	XXXX			andis.	P1		SR	97	AND Immediate Shifted D-form
010010	I	XXXX			b[l][a]	P1			40	Branch I-form
010000	I	XXXX			bc[l][a]	P1		CT	40	Branch Conditional B-form
010011 .... ///.. 10000 10000.	I	XXXX			bcctr[l]	P1		CT	41	Branch Conditional to Count Register XL-form
010011 .... ///.. 00000 10000.	I	XXXX			bclr[l]	P1		CT	41	Branch Conditional to Link Register XL-form
011111 .../. 00000 00000/	I	XXXX			cmp	P1			90	Compare X-form
001011 .../.	I	XXXX			cmpi	P1			90	Compare Immediate D-form
011111 .../. 00001 00000/	I	XXXX			cmpl	P1			91	Compare Logical X-form
001010 .../.	I	XXXX			cmpli	P1			91	Compare Logical Immediate D-form
011111 .... ///// 00000 11010.	I	XXXX			cntlzw[.]	P1		SR	101	Count Leading Zeros Word X-form
010011 .... 01000 00001/	I	XXXX			crand	P1			43	Condition Register AND XL-form
010011 .... 00100 00001/	I	XXXX			crandc	P1			44	Condition Register AND with Complement XL-form
010011 .... 01001 00001/	I	XXXX			creqv	P1			44	Condition Register Equivalent XL-form
010011 .... 00111 00001/	I	XXXX			crnand	P1			43	Condition Register NAND XL-form
010011 .... 00001 00001/	I	XXXX			crnor	P1			44	Condition Register NOR XL-form
010011 .... 01110 00001/	I	XXXX			cror	P1			43	Condition Register OR XL-form
010011 .... 01101 00001/	I	XXXX			crorc	P1			44	Condition Register OR with Complement XL-form
010011 .... 00110 00001/	I	XXXX			crxor	P1			43	Condition Register XOR XL-form
011111 ///// 11111 10110/	II	XXXX			dcbz	P1			1001	Data Cache Block set to Zero X-form
011111 .... 01000 11100.	I	XXXX			eqv[.]	P1		SR	100	Equivalent X-form
011111 .... ///// 11100 11010.	I	XXXX			extsh[.]	P1		SR	101	Extend Sign Halfword X-form
111111 ..... ///// ..... 01000 01000.	I	.XXX			fabs[.]	P1			160	Floating Absolute Value X-form
111111 .... ///// 10101.	I	.XXX			fadd[.]	P1			162	Floating Add A-form
111111 ...// 00001 00000/	I	.XXX			fcmpo	P1			179	Floating Compare Ordered X-form
111111 ...// 00000 00000/	I	.XXX			fcmpu	P1			179	Floating Compare Unordered X-form
111111 .... ///// 10010.	I	.XXX			fdiv[.]	P1			163	Floating Divide A-form
111111 .... 11101.	I	.XXX			fmadd[.]	P1			168	Floating Multiply-Add A-form
111111 ..... ///// ..... 00010 01000.	I	.XXX			fmr[.]	P1			160	Floating Move Register X-form
111111 .... 11100.	I	.XXX			fmsub[.]	P1			168	Floating Multiply-Subtract A-form
111111 .... ///// ..... 11001.	I	.XXX			fmul[.]	P1			163	Floating Multiply A-form
111111 ..... ///// ..... 00100 01000.	I	.XXX			fnabs[.]	P1			160	Floating Negative Absolute Value X-form
111111 ..... ///// ..... 00001 01000.	I	.XXX			fneg[.]	P1			160	Floating Negate X-form
111111 .... 11111.	I	.XXX			fnmadd[.]	P1			169	Floating Negative Multiply-Add A-form
111111 .... 11110.	I	.XXX			fnmsub[.]	P1			169	Floating Negative Multiply-Subtract A-form
111111 ..... ///// ..... 00000 01100.	I	.XXX			frsp[.]	P1			170	Floating Round to Single-Precision X-form
111111 .... ///// 10100.	I	.XXX			fsub[.]	P1			162	Floating Subtract A-form
010011 ///// ///// ///// 00100 10110/	II	XXXX			isync	P1			1014	Instruction Synchronize XL-form
100010	I	XXXX			lbz	P1			48	Load Byte and Zero D-form
100011	I	XXXX			lbzu	P1			49	Load Byte and Zero with Update D-form

Table F.1: Power ISA Instruction Set Sorted by Version (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
011111 .... 00011 10111/	I	XXXX			lbzux	P1			49	Load Byte and Zero with Update Indexed X-form
011111 .... 00010 10111/	I	XXXX			lbzx	P1			48	Load Byte and Zero Indexed X-form
110010	I	.XXX			lfd	P1			152	Load Floating-Point Double D-form
110011	I	.XXX			lfdu	P1			153	Load Floating-Point Double with Update D-form
011111 .... 10011 10111/	I	.XXX			lfdux	P1			153	Load Floating-Point Double with Update Indexed X-form
011111 .... 10010 10111/	I	.XXX			lfdx	P1			152	Load Floating-Point Double Indexed X-form
110000	I	.XXX			lfs	P1			150	Load Floating-Point Single D-form
110001	I	.XXX			lfsu	P1			151	Load Floating-Point Single with Update D-form
011111 .... 10001 10111/	I	.XXX			lfsux	P1			151	Load Floating-Point Single with Update Indexed X-form
011111 .... 10000 10111/	I	.XXX			lfsx	P1			150	Load Floating-Point Single Indexed X-form
101010	I	XXXX			lha	P1			51	Load Halfword Algebraic D-form
101011	I	XXXX			lhau	P1			52	Load Halfword Algebraic with Update D-form
011111 .... 01011 10111/	I	XXXX			lhaux	P1			52	Load Halfword Algebraic with Update Indexed X-form
011111 .... 01010 10111/	I	XXXX			lhax	P1			52	Load Halfword Algebraic Indexed X-form
011111 .... 11000 10110/	I	XXXX			lhbrx	P1			66	Load Halfword Byte-Reverse Indexed X-form
101000	I	XXXX			lhz	P1			50	Load Halfword and Zero D-form
101001	I	XXXX			lhzu	P1			51	Load Halfword and Zero with Update D-form
011111 .... 01001 10111/	I	XXXX			lhzux	P1			51	Load Halfword and Zero with Update Indexed X-form
011111 .... 01000 10111/	I	XXXX			lhzx	P1			50	Load Halfword and Zero Indexed X-form
101110	I	...X			lmw	P1			68	Load Multiple Word D-form
011111 .... 10010 10101/	I	...X			lswi	P1			70	Load String Word Immediate X-form
011111 .... 10000 10101/	I	...X			lswx	P1			70	Load String Word Indexed X-form
011111 .... 10000 10110/	I	XXXX			lwbrx	P1			66	Load Word Byte-Reverse Indexed X-form
100000	I	XXXX			lwz	P1			53	Load Word and Zero D-form
100001	I	XXXX			lwzu	P1			53	Load Word and Zero with Update D-form
011111 .... 00001 10111/	I	XXXX			lwzux	P1			53	Load Word and Zero with Update Indexed X-form
011111 .... 00000 10111/	I	XXXX			lwzx	P1			53	Load Word and Zero Indexed X-form
010011 ...// ...// ///// 00000 00000/	I	XXXX			mcrf	P1			45	Move Condition Register Field XL-form
111111 ...// ...// ///// 00010 00000/	I	.XXX			mcrfs	P1			184	Move to Condition Register from FPSCR X-form
011111 ..... 0//// ///// 00000 10011/	I	XXXX			mfcr	P1			130	Move From Condition Register XFX-form
111111 ..... 00000 ///// 10010 00111.	I	.XXX			mffs[.]	P1			182	Move From FPSCR X-form
011111 ..... ///// ///// 00010 10011/	III	XXXX			mfmsr	P1	P		1110	Move From Machine State Register X-form
011111 .... 01010 10011/	I III	XXXX			mfspr	P1	O		128	Move From Special Purpose Register XFX-form
011111 ..... 0.... ..../ 00100 10000/	I	XXXX			mtcrf	P1			129	Move To Condition Register Fields XFX-form
111111 ..... ///// ///// 00010 00110.	I	.XXX			mtfsb0[.]	P1			185	Move To FPSCR Bit 0 X-form
111111 ..... ///// ///// 00001 00110.	I	.XXX			mtfsb1[.]	P1			185	Move To FPSCR Bit 1 X-form
111111 .... 10110 00111.	I	.XXX			mtfsf[.]	P1			185	Move To FPSCR Fields XFL-form
111111 ...// ////. ..../ 00100 00110.	I	.XXX			mtfsfi[.]	P1			184	Move To FPSCR Field Immediate X-form
011111 ..... ////. ///// 00100 10010/	III	XXXX			mtmsr	P1	P		1108	Move To Machine State Register X-form
011111 .... 01110 10011/	I III	XXXX			mtspr	P1	O		126	Move To Special Purpose Register XFX-form
000111	I	XXXX			mulli	P1			79	Multiply Low Immediate D-form
011111 .... 00111 01011.	I	XXXX			mullw[.]	P1		SR	79	Multiply Low Word XO-form
011111 .... 10111 01011.	I	..XX			mullwo[.]	P1		SR	79	Multiply Low Word & record OV XO-form
011111 .... 01110 11100.	I	XXXX			nand[.]	P1		SR	99	NAND X-form
011111 .... ///// 00011 01000.	I	XXXX			neg[.]	P1		SR	78	Negate XO-form
011111 .... ///// 10011 01000.	I	..XX			nego[.]	P1		SR	78	Negate & record OV XO-form
011111 .... 00011 11100.	I	XXXX			nor[.]	P1		SR	100	NOR X-form

Table F.1: Power ISA Instruction Set Sorted by Version (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
011111 .... 01101 11100.	I	XXXX			or[.]	P1		SR	99	OR X-form
011111 .... 01100 11100.	I	XXXX			orc[.]	P1		SR	100	OR with Complement X-form
011000	I	XXXX			ori	P1			98	OR Immediate D-form
011001	I	XXXX			oris	P1			98	OR Immediate Shifted D-form
010100	I	XXXX			rlwimi[.]	P1		SR	109	Rotate Left Word Immediate then Mask Insert M-form
010101	I	XXXX			rlwinm[.]	P1		SR	108	Rotate Left Word Immediate then AND with Mask M-form
010111	I	XXXX			rlwnm[.]	P1		SR	109	Rotate Left Word then AND with Mask M-form
011111 .... 00000 11000.	I	XXXX			slw[.]	P1		SR	113	Shift Left Word X-form
011111 .... 11000 11000.	I	XXXX			sraw[.]	P1		SR	114	Shift Right Algebraic Word X-form
011111 .... 11001 11000.	I	XXXX			srawi[.]	P1		SR	114	Shift Right Algebraic Word Immediate X-form
011111 .... 10000 11000.	I	XXXX			srw[.]	P1		SR	113	Shift Right Word X-form
100110	I	XXXX			stb	P1			57	Store Byte D-form
100111	I	XXXX			stbu	P1			57	Store Byte with Update D-form
011111 .... 00111 10111/	I	XXXX			stbux	P1			57	Store Byte with Update Indexed X-form
011111 .... 00110 10111/	I	XXXX			stbx	P1			57	Store Byte Indexed X-form
110110	I	.XXX			stfd	P1			156	Store Floating-Point Double D-form
110111	I	.XXX			stfdu	P1			156	Store Floating-Point Double with Update D-form
011111 .... 10111 10111/	I	.XXX			stfdux	P1			157	Store Floating-Point Double with Update Indexed X-form
011111 .... 10110 10111/	I	.XXX			stfdx	P1			156	Store Floating-Point Double Indexed X-form
110100	I	.XXX			stfs	P1			154	Store Floating-Point Single D-form
110101	I	.XXX			stfsu	P1			155	Store Floating-Point Single with Update D-form
011111 .... 10101 10111/	I	.XXX			stfsux	P1			155	Store Floating-Point Single with Update Indexed X-form
011111 .... 10100 10111/	I	.XXX			stfsx	P1			155	Store Floating-Point Single Indexed X-form
101100	I	XXXX			sth	P1			58	Store Halfword D-form
011111 .... 11100 10110/	I	XXXX			sthbrx	P1			66	Store Halfword Byte-Reverse Indexed X-form
101101	I	XXXX			sthu	P1			58	Store Halfword with Update D-form
011111 .... 01101 10111/	I	XXXX			sthux	P1			58	Store Halfword with Update Indexed X-form
011111 .... 01100 10111/	I	XXXX			sthx	P1			58	Store Halfword Indexed X-form
101111	I	...X			stmw	P1			68	Store Multiple Word D-form
011111 .... 10110 10101/	I	...X			stswi	P1			71	Store String Word Immediate X-form
011111 .... 10100 10101/	I	...X			stswx	P1			71	Store String Word Indexed X-form
100100	I	XXXX			stw	P1			59	Store Word D-form
011111 .... 10100 10110/	I	XXXX			stwbrx	P1			66	Store Word Byte-Reverse Indexed X-form
100101	I	XXXX			stwu	P1			59	Store Word with Update D-form
011111 .... 00101 10111/	I	XXXX			stwux	P1			59	Store Word with Update Indexed X-form
011111 .... 00100 10111/	I	XXXX			stwx	P1			59	Store Word Indexed X-form
011111 .... 00000 01000.	I	XXXX			subfc[.]	P1		SR	76	Subtract From Carrying XO-form
011111 .... 10000 01000.	I	..XX			subfco[.]	P1		SR	76	Subtract From Carrying & record OV XO-form
011111 .... 00100 01000.	I	XXXX			subfe[.]	P1		SR	76	Subtract From Extended XO-form
011111 .... 10100 01000.	I	..XX			subfeo[.]	P1		SR	76	Subtract From Extended & record OV XO-form
001000	I	XXXX			subfic	P1		SR	75	Subtract From Immediate Carrying D-form
011111 .... ///// 00111 01000.	I	XXXX			subfme[.]	P1		SR	77	Subtract From Minus One Extended XO-form
011111 .... ///// 10111 01000.	I	..XX			subfmeo[.]	P1		SR	77	Subtract From Minus One Extended & record OV XO-form
011111 .... ///// 00110 01000.	I	XXXX			subfze[.]	P1		SR	77	Subtract From Zero Extended XO-form
011111 .... ///// 10110 01000.	I	..XX			subfzeo[.]	P1		SR	77	Subtract From Zero Extended & record OV XO-form
011111 //... ///.. ///// 10010 10110/	II	XXXX			sync	P1			1025	Synchronize X-form
011111 ..... /.... ..... 01001 10010/	III	...X			tlbie	P1	HV	64	1174	TLB Invalidate Entry X-form
011111 .... 00000 00100/	I	XXXX			tw	P1			94	Trap Word X-form

Table F.1: Power ISA Instruction Set Sorted by Version (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
000011	I	XXXX			twi	P1			94	Trap Word Immediate D-form
011111 .... 01001 11100.	I	XXXX			xor[.]	P1		SR	99	XOR X-form
011010	I	XXXX			xori	P1			98	XOR Immediate D-form
011011	I	XXXX			xoris	P1			98	XOR Immediate Shifted D-form

Appendix G. Power ISA Instruction Set Sorted by OpenPOWER Compliancy Subset

This appendix lists all the instructions in the Power ISA, sorted by the instruction's OpenPOWER compliancy subset membership, then by mnemonic. For an explanation of the values in the columns, see Appendix E.

Table G.1: Power ISA Instruction Set Sorted by Compliancy Subset

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
011111 .... 01000 01010.	I	XXXX			add[.]	P1		SR	74	Add XO-form
011111 .... 00000 01010.	I	XXXX			addc[.]	P1		SR	76	Add Carrying XO-form
011111 .... 00100 01010.	I	XXXX			adde[.]	P1		SR	76	Add Extended XO-form
011111 ....101 01010/	I	XXXX			addex	v3.0B			78	Add Extended using alternate carry bit Z23-form
011111 .... /0010 01010/	I	XXXX			addg6s	v2.06			117	Add and Generate Sixes XO-form
001110	I	XXXX			addi	P1			73	Add Immediate D-form
001100	I	XXXX			addic	P1		SR	75	Add Immediate Carrying D-form
001101	I	XXXX			addic.	P1		SR	75	Add Immediate Carrying and Record D-form
001111	I	XXXX			addis	P1			74	Add Immediate Shifted D-form
011111 .... ///// 00111 01010.	I	XXXX			addme[.]	P1		SR	77	Add to Minus One Extended XO-form
010011 .... 00010.	I	XXXX			addpcis	v3.0			74	Add PC Immediate Shifted DX-form
011111 .... ///// 00110 01010.	I	XXXX			addze[.]	P1		SR	77	Add to Zero Extended XO-form
011111 .... 00000 11100.	I	XXXX			and[.]	P1		SR	99	AND X-form
011111 .... 00001 11100.	I	XXXX			andc[.]	P1		SR	100	AND with Complement X-form
011100	I	XXXX			andi.	P1		SR	97	AND Immediate D-form
011101	I	XXXX			andis.	P1		SR	97	AND Immediate Shifted D-form
010010	I	XXXX			b[l][a]	P1			40	Branch I-form
010000	I	XXXX			bc[l][a]	P1		CT	40	Branch Conditional B-form
010011 .... ///.. 10000 10000.	I	XXXX			bcctr[l]	P1		CT	41	Branch Conditional to Count Register XL-form
010011 .... ///.. 00000 10000.	I	XXXX			bclr[l]	P1		CT	41	Branch Conditional to Link Register XL-form
010011 .... ///.. 10001 10000.	I	XXXX			bctar[l]	v2.07			42	Branch Conditional to Branch Target Address Register XL-form
011111 .... ///// 00110 11011/	I	XXXX			brh	v3.1			118	Byte-Reverse Halfword X-form
011111 .... ///// 00100 11011/	I	XXXX			brw	v3.1			118	Byte-Reverse Word X-form
011111 .... ///// 01001 11010/	I	XXXX			cbcdtd	v2.06			117	Convert Binary Coded Decimal To Declets X-form
011111 .... ///// 01000 11010/	I	XXXX			cdtbcd	v2.06			117	Convert Declets To Binary Coded Decimal X-form
011111 .../. 00000 00000/	I	XXXX			cmp	P1			90	Compare X-form
011111 .... 01111 11100/	I	XXXX			cmpb	v2.05			101	Compare Bytes X-form
011111 ...// 00111 00000/	I	XXXX			cmpeqb	v3.0			93	Compare Equal Byte X-form
001011 .../.	I	XXXX			cmpi	P1			90	Compare Immediate D-form
011111 .../. 00001 00000/	I	XXXX			cmpl	P1			91	Compare Logical X-form

Table G.1: Power ISA Instruction Set Sorted by Compliancy Subset (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
001010 …/.	I	XXXX			cmpli	P1			91	Compare Logical Immediate D-form
011111 …/. 00110 00000/	I	XXXX			cmprb	v3.0			92	Compare Ranged Byte X-form
011111 .... ///// 00000 11010.	I	XXXX			cntlzw[.]	P1		SR	101	Count Leading Zeros Word X-form
011111 .... ///// 10000 11010.	I	XXXX			cnttzw[.]	v3.0			101	Count Trailing Zeros Word X-form
010011 .... 01000 00001/	I	XXXX			crand	P1			43	Condition Register AND XL-form
010011 .... 00100 00001/	I	XXXX			crandc	P1			44	Condition Register AND with Complement XL-form
010011 .... 01001 00001/	I	XXXX			creqv	P1			44	Condition Register Equivalent XL-form
010011 .... 00111 00001/	I	XXXX			crnand	P1			43	Condition Register NAND XL-form
010011 .... 00001 00001/	I	XXXX			crnor	P1			44	Condition Register NOR XL-form
010011 .... 01110 00001/	I	XXXX			cror	P1			43	Condition Register OR XL-form
010011 .... 01101 00001/	I	XXXX			crorc	P1			44	Condition Register OR with Complement XL-form
010011 .... 00110 00001/	I	XXXX			crxor	P1			43	Condition Register XOR XL-form
011111 ..... ///.. ///// 10111 10011/	I	XXXX			darn	v3.0			84	Deliver A Random Number X-form
011111 ///// 11111 10110/	II	XXXX			dcbz	P1			1001	Data Cache Block set to Zero X-form
011111 .... 01111 01011.	I	XXXX			divw[.]	PPC		SR	80	Divide Word XO-form
011111 .... 01101 01011.	I	XXXX			divwe[.]	v2.06		SR	81	Divide Word Extended XO-form
011111 .... 01100 01011.	I	XXXX			divweu[.]	v2.06		SR	81	Divide Word Extended Unsigned XO-form
011111 .... 01110 01011.	I	XXXX			divwu[.]	PPC		SR	80	Divide Word Unsigned XO-form
011111 ///// ///// ///// 11010 10110/	II	XXXX			eieio	PPC			1028	Enforce In-order Execution of I/O X-form
011111 .... 01000 11100.	I	XXXX			eqv[.]	P1		SR	100	Equivalent X-form
011111 .... ///// 11101 11010.	I	XXXX			extsb[.]	PPC		SR	101	Extend Sign Byte X-form
011111 .... ///// 11100 11010.	I	XXXX			extsh[.]	P1		SR	101	Extend Sign Halfword X-form
011111 .... 01111/	I	XXXX			isel	v2.03			96	Integer Select A-form
010011 ///// ///// ///// 00100 10110/	II	XXXX			isync	P1			1014	Instruction Synchronize XL-form
011111 .... 00001 10100.	II	XXXX			lbarx	v2.06			1016	Load Byte And Reserve Indexed X-form
100010	I	XXXX			lbz	P1			48	Load Byte and Zero D-form
100011	I	XXXX			lbzu	P1			49	Load Byte and Zero with Update D-form
011111 .... 00011 10111/	I	XXXX			lbzux	P1			49	Load Byte and Zero with Update Indexed X-form
011111 .... 00010 10111/	I	XXXX			lbzx	P1			48	Load Byte and Zero Indexed X-form
101010	I	XXXX			lha	P1			51	Load Halfword Algebraic D-form
011111 .... 00011 10100.	II	XXXX			lharx	v2.06			1016	Load Halfword And Reserve Indexed X-form
101011	I	XXXX			lhau	P1			52	Load Halfword Algebraic with Update D-form
011111 .... 01011 10111/	I	XXXX			lhaux	P1			52	Load Halfword Algebraic with Update Indexed X-form
011111 .... 01010 10111/	I	XXXX			lhax	P1			52	Load Halfword Algebraic Indexed X-form
011111 .... 11000 10110/	I	XXXX			lhbrx	P1			66	Load Halfword Byte-Reverse Indexed X-form
101000	I	XXXX			lhz	P1			50	Load Halfword and Zero D-form
101001	I	XXXX			lhzu	P1			51	Load Halfword and Zero with Update D-form
011111 .... 01001 10111/	I	XXXX			lhzux	P1			51	Load Halfword and Zero with Update Indexed X-form
011111 .... 01000 10111/	I	XXXX			lhzx	P1			50	Load Halfword and Zero Indexed X-form
011111 .... 00000 10100.	II	XXXX			lwarx	PPC			1017	Load Word And Reserve Indexed X-form
011111 .... 10000 10110/	I	XXXX			lwbrx	P1			66	Load Word Byte-Reverse Indexed X-form
100000	I	XXXX			lwz	P1			53	Load Word and Zero D-form
100001	I	XXXX			lwzu	P1			53	Load Word and Zero with Update D-form
011111 .... 00001 10111/	I	XXXX			lwzux	P1			53	Load Word and Zero with Update Indexed X-form
011111 .... 00000 10111/	I	XXXX			lwzx	P1			53	Load Word and Zero Indexed X-form
010011 …// …// ///// 00000 00000/	I	XXXX			mcrf	P1			45	Move Condition Register Field XL-form
011111 …// ///// ///// 10010 00000/	I	XXXX			mcrxrx	v3.0			129	Move to CR from XER Extended X-form
011111 ..... 0//// ///// 00000 10011/	I	XXXX			mfcr	P1			130	Move From Condition Register XFX-form

Table G.1: Power ISA Instruction Set Sorted by Compliancy Subset (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
011111 ..... ///// ///// 00010 10011/	III	XXXX			mfmsr	P1	P		1110	Move From Machine State Register X-form
011111 ..... 1.... ..../ 00000 10011/	I	XXXX			mfocrf	v2.01			130	Move From One Condition Register Field XFX-form
011111 .... 01010 10011/	I III	XXXX			mfspr	P1	O		128	Move From Special Purpose Register XFX-form
011111 .... 01011 10011/	II	XXXX			mftb	PPC			1032	Move From Time Base XFX-form
011111 .... 11000 01011/	I	XXXX			modsw	v3.0			83	Modulo Signed Word X-form
011111 .... 01000 01011/	I	XXXX			moduw	v3.0			83	Modulo Unsigned Word X-form
011111 ..... 0.... ..../ 00100 10000/	I	XXXX			mtcrf	P1			129	Move To Condition Register Fields XFX-form
011111 ..... ////. ///// 00100 10010/	III	XXXX			mtmsr	P1	P		1108	Move To Machine State Register X-form
011111 ..... 1.... ..../ 00100 10000/	I	XXXX			mtocrf	v2.01			129	Move To One Condition Register Field XFX-form
011111 .... 01110 10011/	I III	XXXX			mtspr	P1	O		126	Move To Special Purpose Register XFX-form
011111 .... /0010 01011.	I	XXXX			mulhw[.]	PPC		SR	79	Multiply High Word XO-form
011111 .... /0000 01011.	I	XXXX			mulhwu[.]	PPC		SR	79	Multiply High Word Unsigned XO-form
000111	I	XXXX			mulli	P1			79	Multiply Low Immediate D-form
011111 .... 00111 01011.	I	XXXX			mullw[.]	P1		SR	79	Multiply Low Word XO-form
011111 .... 01110 11100.	I	XXXX			nand[.]	P1		SR	99	NAND X-form
011111 .... ///// 00011 01000.	I	XXXX			neg[.]	P1		SR	78	Negate XO-form
011111 .... 00011 11100.	I	XXXX			nor[.]	P1		SR	100	NOR X-form
011111 .... 01101 11100.	I	XXXX			or[.]	P1		SR	99	OR X-form
011111 .... 01100 11100.	I	XXXX			orc[.]	P1		SR	100	OR with Complement X-form
011000	I	XXXX			ori	P1			98	OR Immediate D-form
011001	I	XXXX			oris	P1			98	OR Immediate Shifted D-form
011111 .... ///// 00011 11010/	I	XXXX			popcntb	v2.02			102	Population Count Bytes X-form
011111 .... ///// 01011 11010/	I	XXXX			popcntw	v2.06			102	Population Count Words X-form
011111 .... ///// 00100 11010/	I	XXXX			prtyw	v2.05			102	Parity Word X-form
010011 ///// ///// ///// 00000 10010/	III	XXXX			rfid	PPC	P		1086	Return From Interrupt Doubleword XL-form
010011 ///// ///// ///// 00010 10010/	III	XXXX			rfscv	v3.0	P		1085	Return From System Call Vectored XL-form
010100	I	XXXX			rlwimi[.]	P1		SR	109	Rotate Left Word Immediate then Mask Insert M-form
010101	I	XXXX			rlwinm[.]	P1		SR	108	Rotate Left Word Immediate then AND with Mask M-form
010111	I	XXXX			rlwnm[.]	P1		SR	109	Rotate Left Word then AND with Mask M-form
010001 ///// ///// ////. ..... .///1/	I	XXXX			sc	PPC			45	System Call SC-form
010001 ///// ///// ////. ..... .///01	I	XXXX			scv	v3.0			45	System Call Vectored SC-form
011111 ....// ///// 00100 00000/	I	XXXX			setb	v3.0			131	Set Boolean X-form
011111 .... ///// 01100 00000/	I	XXXX			setbc	v3.1			131	Set Boolean Condition X-form
011111 .... ///// 01101 00000/	I	XXXX			setbcr	v3.1			131	Set Boolean Condition Reverse X-form
011111 .... ///// 01110 00000/	I	XXXX			setnbc	v3.1			131	Set Negative Boolean Condition X-form
011111 .... ///// 01111 00000/	I	XXXX			setnbcr	v3.1			131	Set Negative Boolean Condition Reverse X-form
011111 .... 00000 11000.	I	XXXX			slw[.]	P1		SR	113	Shift Left Word X-form
011111 .... 11000 11000.	I	XXXX			sraw[.]	P1		SR	114	Shift Right Algebraic Word X-form
011111 .... 11001 11000.	I	XXXX			srawi[.]	P1		SR	114	Shift Right Algebraic Word Immediate X-form
011111 .... 10000 11000.	I	XXXX			srw[.]	P1		SR	113	Shift Right Word X-form
100110	I	XXXX			stb	P1			57	Store Byte D-form
011111 .... 10101 101101	II	XXXX			stbcx.	v2.06			1018	Store Byte Conditional Indexed X-form
100111	I	XXXX			stbu	P1			57	Store Byte with Update D-form
011111 .... 00111 10111/	I	XXXX			stbux	P1			57	Store Byte with Update Indexed X-form
011111 .... 00110 10111/	I	XXXX			stbx	P1			57	Store Byte Indexed X-form
101100	I	XXXX			sth	P1			58	Store Halfword D-form
011111 .... 11100 10110/	I	XXXX			sthbrx	P1			66	Store Halfword Byte-Reverse Indexed X-form

Table G.1: Power ISA Instruction Set Sorted by Compliancy Subset (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
011111 .... 10110 101101	II	XXXX			sthcx.	v2.06			1019	Store Halfword Conditional Indexed X-form
101101	I	XXXX			sthu	P1			58	Store Halfword with Update D-form
011111 .... 01101 10111/	I	XXXX			sthux	P1			58	Store Halfword with Update Indexed X-form
011111 .... 01100 10111/	I	XXXX			sthx	P1			58	Store Halfword Indexed X-form
100100	I	XXXX			stw	P1			59	Store Word D-form
011111 .... 10100 10110/	I	XXXX			stwbrx	P1			66	Store Word Byte-Reverse Indexed X-form
011111 .... 00100 101101	II	XXXX			stwcx.	PPC			1020	Store Word Conditional Indexed X-form
100101	I	XXXX			stwu	P1			59	Store Word with Update D-form
011111 .... 00101 10111/	I	XXXX			stwux	P1			59	Store Word with Update Indexed X-form
011111 .... 00100 10111/	I	XXXX			stwx	P1			59	Store Word Indexed X-form
011111 .... 00001 01000.	I	XXXX			subf[.]	PPC		SR	75	Subtract From XO-form
011111 .... 00000 01000.	I	XXXX			subfc[.]	P1		SR	76	Subtract From Carrying XO-form
011111 .... 00100 01000.	I	XXXX			subfe[.]	P1		SR	76	Subtract From Extended XO-form
001000	I	XXXX			subfic	P1		SR	75	Subtract From Immediate Carrying D-form
011111 .... ///// 00111 01000.	I	XXXX			subfme[.]	P1		SR	77	Subtract From Minus One Extended XO-form
011111 .... ///// 00110 01000.	I	XXXX			subfze[.]	P1		SR	77	Subtract From Zero Extended XO-form
011111 //... ///.. ///// 10010 10110/	II	XXXX			sync	P1			1025	Synchronize X-form
011111 .... 00000 00100/	I	XXXX			tw	P1			94	Trap Word X-form
000011	I	XXXX			twi	P1			94	Trap Word Immediate D-form
011111 .... 01001 11100.	I	XXXX			xor[.]	P1		SR	99	XOR X-form
011010	I	XXXX			xori	P1			98	XOR Immediate D-form
011011	I	XXXX			xoris	P1			98	XOR Immediate Shifted D-form
111111 ..... ///// ..... 01000 01000.	I	.XXX			fabs[.]	P1			160	Floating Absolute Value X-form
111111 .... ///// 10101.	I	.XXX			fadd[.]	P1			162	Floating Add A-form
111011 .... ///// 10101.	I	.XXX			fadds[.]	PPC			162	Floating Add Single A-form
111111 ..... ///// ..... 11010 01110.	I	.XXX			fcfid[.]	PPC			175	Floating Convert with round Signed Doubleword to Double-Precision format X-form
111011 ..... ///// ..... 11010 01110.	I	.XXX			fcfids[.]	v2.06			176	Floating Convert with round Signed Doubleword to Single-Precision format X-form
111111 ..... ///// ..... 11110 01110.	I	.XXX			fcfidu[.]	v2.06			175	Floating Convert with round Unsigned Doubleword to Double-Precision format X-form
111011 ..... ///// ..... 11110 01110.	I	.XXX			fcfidus[.]	v2.06			176	Floating Convert with round Unsigned Doubleword to Single-Precision format X-form
111111 ...// 00001 00000/	I	.XXX			fcmpo	P1			179	Floating Compare Ordered X-form
111111 ...// 00000 00000/	I	.XXX			fcmpu	P1			179	Floating Compare Unordered X-form
111111 .... 00000 01000.	I	.XXX			fcpsgn[.]	v2.05			160	Floating Copy Sign X-form
111111 ..... ///// ..... 11001 01110.	I	.XXX			fctid[.]	PPC			171	Floating Convert with round Double-Precision To Signed Doubleword format X-form
111111 ..... ///// ..... 11101 01110.	I	.XXX			fctidu[.]	v2.06			172	Floating Convert with round Double-Precision To Unsigned Doubleword format X-form
111111 ..... ///// ..... 11101 01111.	I	.XXX			fctiduz[.]	v2.06			172	Floating Convert with truncate Double-Precision To Unsigned Doubleword format X-form
111111 ..... ///// ..... 11001 01111.	I	.XXX			fctidz[.]	PPC			171	Floating Convert with truncate Double-Precision To Signed Doubleword format X-form
111111 ..... ///// ..... 00000 01110.	I	.XXX			fctiw[.]	P2			173	Floating Convert with round Double-Precision To Signed Word format X-form
111111 ..... ///// ..... 00100 01110.	I	.XXX			fctiwu[.]	v2.06			174	Floating Convert with round Double-Precision To Unsigned Word format X-form

Table G.1: Power ISA Instruction Set Sorted by Compliancy Subset (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
111111 ..... ///// ..... 00100 01111.	I	.XXX			fctiwuz[.]	v2.06			174	Floating Convert with truncate Double-Precision To Unsigned Word format X-form
111111 ..... ///// ..... 00000 01111.	I	.XXX			fctiwz[.]	P2			173	Floating Convert with truncate Double-Precision To Signed Word format X-form
111111 .... ///// 10010.	I	.XXX			fdiv[.]	P1			163	Floating Divide A-form
111011 .... ///// 10010.	I	.XXX			fdivs[.]	PPC			163	Floating Divide Single A-form
111111 .... 11101.	I	.XXX			fmadd[.]	P1			168	Floating Multiply-Add A-form
111011 .... 11101.	I	.XXX			fmadds[.]	PPC			168	Floating Multiply-Add Single A-form
111111 ..... ///// ..... 00010 01000.	I	.XXX			fmr[.]	P1			160	Floating Move Register X-form
111111 .... 11110 00110/	I	.XXX			fmrgew	v2.07			161	Floating Merge Even Word X-form
111111 .... 11010 00110/	I	.XXX			fmrgow	v2.07			161	Floating Merge Odd Word X-form
111111 .... 11100.	I	.XXX			fmsub[.]	P1			168	Floating Multiply-Subtract A-form
111011 .... 11100.	I	.XXX			fmsubs[.]	PPC			168	Floating Multiply-Subtract Single A-form
111111 .... ///// ..... 11001.	I	.XXX			fmul[.]	P1			163	Floating Multiply A-form
111011 .... ///// ..... 11001.	I	.XXX			fmuls[.]	PPC			163	Floating Multiply Single A-form
111111 ..... ///// ..... 00100 01000.	I	.XXX			fnabs[.]	P1			160	Floating Negative Absolute Value X-form
111111 ..... ///// ..... 00001 01000.	I	.XXX			fneg[.]	P1			160	Floating Negate X-form
111111 .... 11111.	I	.XXX			fnmadd[.]	P1			169	Floating Negative Multiply-Add A-form
111011 .... 11111.	I	.XXX			fnmadds[.]	PPC			169	Floating Negative Multiply-Add Single A-form
111111 .... 11110.	I	.XXX			fnmsub[.]	P1			169	Floating Negative Multiply-Subtract A-form
111011 .... 11110.	I	.XXX			fnmsubs[.]	PPC			169	Floating Negative Multiply-Subtract Single A-form
111111 ..... ///// ..... ///// 11000.	I	.XXX			fre[.]	v2.02			165	Floating Reciprocal Estimate A-form
111011 ..... ///// ..... ///// 11000.	I	.XXX			fres[.]	PPC			165	Floating Reciprocal Estimate Single A-form
111111 ..... ///// ..... 01111 01000.	I	.XXX			frim[.]	v2.02			178	Floating Round to Integer Minus X-form
111111 ..... ///// ..... 01100 01000.	I	.XXX			frin[.]	v2.02			177	Floating Round to Integer Nearest X-form
111111 ..... ///// ..... 01110 01000.	I	.XXX			frip[.]	v2.02			178	Floating Round to Integer Plus X-form
111111 ..... ///// ..... 01101 01000.	I	.XXX			friz[.]	v2.02			177	Floating Round to Integer Toward Zero X-form
111111 ..... ///// ..... 00000 01100.	I	.XXX			frsp[.]	P1			170	Floating Round to Single-Precision X-form
111111 ..... ///// ..... ///// 11010.	I	.XXX			frsqrte[.]	PPC			166	Floating Reciprocal Square Root Estimate A-form
111011 ..... ///// ..... ///// 11010.	I	.XXX			frsqrtes[.]	v2.02			166	Floating Reciprocal Square Root Estimate Single A-form
111111 .... 10111.	I	.XXX			fsel[.]	PPC			180	Floating Select A-form
111111 ..... ///// ..... ///// 10110.	I	.XXX			fsqrt[.]	P2			164	Floating Square Root A-form
111011 ..... ///// ..... ///// 10110.	I	.XXX			fsqrts[.]	PPC			164	Floating Square Root Single A-form
111111 .... ///// 10100.	I	.XXX			fsub[.]	P1			162	Floating Subtract A-form
111011 .... ///// 10100.	I	.XXX			fsubs[.]	PPC			162	Floating Subtract Single A-form
111111 ...// 00100 00000/	I	.XXX			ftdiv	v2.06			167	Floating Test for software Divide X-form
111111 ...// ///// 00101 00000/	I	.XXX			ftsqrt	v2.06			167	Floating Test for software Square Root X-form
110010	I	.XXX			lfd	P1			152	Load Floating-Point Double D-form
110011	I	.XXX			lfdu	P1			153	Load Floating-Point Double with Update D-form
011111 .... 10011 10111/	I	.XXX			lfdux	P1			153	Load Floating-Point Double with Update Indexed X-form
011111 .... 10010 10111/	I	.XXX			lfdx	P1			152	Load Floating-Point Double Indexed X-form
011111 .... 11010 10111/	I	.XXX			lfiwax	v2.05			153	Load Floating-Point as Integer Word Algebraic Indexed X-form
011111 .... 11011 10111/	I	.XXX			lfiwzx	v2.06			153	Load Floating-Point as Integer Word & Zero Indexed X-form
110000	I	.XXX			lfs	P1			150	Load Floating-Point Single D-form
110001	I	.XXX			lfsu	P1			151	Load Floating-Point Single with Update D-form
011111 .... 10001 10111/	I	.XXX			lfsux	P1			151	Load Floating-Point Single with Update Indexed X-form

Table G.1: Power ISA Instruction Set Sorted by Compliancy Subset (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
011111 .... 10000 10111/	I	.XXX			lfsx	P1			150	Load Floating-Point Single Indexed X-form
111111 ...// ...// ///// 00010 00000/	I	.XXX			mcrfs	P1			184	Move to Condition Register from FPSCR X-form
111111 ..... 00000 ///// 10010 00111.	I	.XXX			mffs[.]	P1			182	Move From FPSCR X-form
111111 ..... 10100 ..... 10010 00111/	I	.XXX			mffscdrn	v3.0B			183	Move From FPSCR Control & Set DRN X-form
111111 ..... 10101 //... 10010 00111/	I	.XXX			mffscdrni	v3.0B			183	Move From FPSCR Control & Set DRN Immediate X-form
111111 ..... 00001 ///// 10010 00111/	I	.XXX			mffsce	v3.0B			182	Move From FPSCR & Clear Enables X-form
111111 ..... 10110 ..... 10010 00111/	I	.XXX			mffscrn	v3.0B			183	Move From FPSCR Control & Set RN X-form
111111 ..... 10111 ///.. 10010 00111/	I	.XXX			mffscrni	v3.0B			183	Move From FPSCR Control & Set RN Immediate X-form
111111 ..... 11000 ///// 10010 00111/	I	.XXX			mffsl	v3.0B			184	Move From FPSCR Lightweight X-form
111111 ..... ///// ///// 00010 00110.	I	.XXX			mtfsb0[.]	P1			185	Move To FPSCR Bit 0 X-form
111111 ..... ///// ///// 00001 00110.	I	.XXX			mtfsb1[.]	P1			185	Move To FPSCR Bit 1 X-form
111111 .... 10110 00111.	I	.XXX			mtfsf[.]	P1			185	Move To FPSCR Fields XFL-form
111111 ...// ////. ..../ 00100 00110.	I	.XXX			mtfsfi[.]	P1			184	Move To FPSCR Field Immediate X-form
110110	I	.XXX			stfd	P1			156	Store Floating-Point Double D-form
110111	I	.XXX			stfdu	P1			156	Store Floating-Point Double with Update D-form
011111 .... 10111 10111/	I	.XXX			stfdux	P1			157	Store Floating-Point Double with Update Indexed X-form
011111 .... 10110 10111/	I	.XXX			stfdx	P1			156	Store Floating-Point Double Indexed X-form
011111 .... 11110 10111/	I	.XXX			stfiwx	PPC			157	Store Floating-Point as Integer Word Indexed X-form
110100	I	.XXX			stfs	P1			154	Store Floating-Point Single D-form
110101	I	.XXX			stfsu	P1			155	Store Floating-Point Single with Update D-form
011111 .... 10101 10111/	I	.XXX			stfsux	P1			155	Store Floating-Point Single with Update Indexed X-form
011111 .... 10100 10111/	I	.XXX			stfsx	P1			155	Store Floating-Point Single Indexed X-form
011111 .... 10000 01010.	I	..XX			addco[.]	P1		SR	76	Add Carrying & record OV XO-form
011111 .... 10100 01010.	I	..XX			addeo[.]	P1		SR	76	Add Extended & record OV XO-form
011111 .... ///// 10111 01010.	I	..XX			addmeo[.]	P1		SR	77	Add to Minus One Extended & record OV XO-form
011111 .... 11000 01010.	I	..XX			addo[.]	P1		SR	74	Add & record OV XO-form
011111 .... ///// 10110 01010.	I	..XX			addzeo[.]	P1		SR	77	Add to Zero Extended & record OV XO-form
000100 .... 1.000 000001	I	..XX			bcdadd.	v2.07			466	Decimal Add Modulo VX-form
000100 ..... 00111 ..... 1.110 000001	I	..XX			bcdcfn.	v3.0			468	Decimal Convert From National VX-form
000100 ..... 00010 ..... 1.110 000001	I	..XX			bcdcfsq.	v3.0			472	Decimal Convert From Signed Quadword VX-form
000100 ..... 00110 ..... 1.110 000001	I	..XX			bcdcfz.	v3.0			469	Decimal Convert From Zoned VX-form
000100 .... 01101 000001	I	..XX			bcdcpsgn.	v3.0			476	Decimal Copy Sign VX-form
000100 ..... 00101 ..... 1/110 000001	I	..XX			bcdctn.	v3.0			470	Decimal Convert To National VX-form
000100 ..... 00000 ..... 1/110 000001	I	..XX			bcdctsq.	v3.0			473	Decimal Convert To Signed Quadword VX-form
000100 ..... 00100 ..... 1.110 000001	I	..XX			bcdctz.	v3.0			471	Decimal Convert To Zoned VX-form
000100 .... 1.011 000001	I	..XX			bcds.	v3.0			478	Decimal Shift VX-form
000100 ..... 11111 ..... 1.110 000001	I	..XX			bcdsetsgn.	v3.0			477	Decimal Set Sign VX-form
000100 .... 1.111 000001	I	..XX			bcdsr.	v3.0			480	Decimal Shift & Round VX-form
000100 .... 1.001 000001	I	..XX			bcdsub.	v2.07			466	Decimal Subtract Modulo VX-form
000100 .... 1.100 000001	I	..XX			bcdtrunc.	v3.0			481	Decimal Truncate VX-form
000100 .... 1/010 000001	I	..XX			bcdus.	v3.0			479	Decimal Unsigned Shift VX-form
000100 .... 1/101 000001	I	..XX			bcdutrunc.	v3.0			482	Decimal Unsigned Truncate VX-form
011111 .... 00111 11100/	I	..XX			bpermd	v2.06			105	Bit Permute Doubleword X-form
011111 .... ///// 00101 11011/	I	..XX			brd	v3.1			118	Byte-Reverse Doubleword X-form
011111 .... 00110 11100/	I	..XX			cfuged	v3.1			105	Centrifuge Doubleword X-form
011111 .... ///// 00001 11010.	I	..XX			cntlzd[.]	PPC		SR	104	Count Leading Zeros Doubleword X-form
011111 .... 00001 11011/	I	..XX			cntlzdm	v3.1			104	Count Leading Zeros Doubleword under bit Mask X-form
011111 .... ///// 10001 11010.	I	..XX			cnttzd[.]	v3.0			104	Count Trailing Zeros Doubleword X-form

Table G.1: Power ISA Instruction Set Sorted by Compliancy Subset (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
011111 .... 10001 11011/	I	..XX			cnttzdm	v3.1			104	Count Trailing Zeros Doubleword under bit Mask X-form
011111 //... .... 00010 10110/	II	..XX			dcbf	PPC			1003	Data Cache Block Flush X-form
011111 ///// 00001 10110/	II	..XX			dcbst	PPC			1002	Data Cache Block Store X-form
011111 .... 01000 10110/	II	..XX			dcbt	PPC			999	Data Cache Block Touch X-form
011111 .... 00111 10110/	II	..XX			dcbtst	PPC			1000	Data Cache Block Touch for Store X-form
011111 .... 01111 01001.	I	..XX			divd[.]	PPC		SR	87	Divide Doubleword XO-form
011111 .... 01101 01001.	I	..XX			divde[.]	v2.06		SR	88	Divide Doubleword Extended XO-form
011111 .... 11101 01001.	I	..XX			divdeo[.]	v2.06		SR	88	Divide Doubleword Extended & record OV XO-form
011111 .... 01100 01001.	I	..XX			divdeu[.]	v2.06		SR	88	Divide Doubleword Extended Unsigned XO-form
011111 .... 11100 01001.	I	..XX			divdeuo[.]	v2.06		SR	88	Divide Doubleword Extended Unsigned & record OV XO-form
011111 .... 11111 01001.	I	..XX			divdo[.]	PPC		SR	87	Divide Doubleword & record OV XO-form
011111 .... 01110 01001.	I	..XX			divdu[.]	PPC		SR	87	Divide Doubleword Unsigned XO-form
011111 .... 11110 01001.	I	..XX			divduo[.]	PPC		SR	87	Divide Doubleword Unsigned & record OV XO-form
011111 .... 11101 01011.	I	..XX			divweo[.]	v2.06		SR	81	Divide Word Extended & record OV XO-form
011111 .... 11100 01011.	I	..XX			divweuo[.]	v2.06		SR	81	Divide Word Extended Unsigned & record OV XO-form
011111 .... 11111 01011.	I	..XX			divwo[.]	PPC		SR	80	Divide Word & record OV XO-form
011111 .... 11110 01011.	I	..XX			divwuo[.]	PPC		SR	80	Divide Word Unsigned & record OV XO-form
011111 .... ///// 11110 11010.	I	..XX			extsw[.]	PPC		SR	103	Extend Sign Word X-form
011111 .... 11011 1101..	I	..XX			extswsli[.]	v3.0			116	Extend Sign Word and Shift Left Immediate XS-form
010011 ///// ///// ///// 01000 10010/	III	..XX			hrfid	v2.02	HV		1086	Hypervisor Return From Interrupt Doubleword XL-form
011111 ///// 11110 10110/	II	..XX			icbi	PPC			991	Instruction Cache Block Invalidate X-form
011111 /.... .... 00000 10110/	II	..XX			icbt	v2.07			991	Instruction Cache Block Touch X-form
011111 .... 11010 10101/	III	..XX			lbzcix	v2.05	HV		1097	Load Byte and Zero Caching Inhibited Indexed X-form
111010 00	I	..XX			ld	PPC			55	Load Doubleword DS-form
011111 .... 00010 10100.	II	..XX			ldarx	PPC			1021	Load Doubleword And Reserve Indexed X-form
011111 .... 10000 10100/	I	..XX			ldbrx	v2.06			67	Load Doubleword Byte-Reverse Indexed X-form
011111 .... 11011 10101/	III	..XX			ldcix	v2.05	HV		1097	Load Doubleword Caching Inhibited Indexed X-form
111010 01	I	..XX			ldu	PPC			55	Load Doubleword with Update DS-form
011111 .... 00001 10101/	I	..XX			ldux	PPC			55	Load Doubleword with Update Indexed X-form
011111 .... 00000 10101/	I	..XX			ldx	PPC			55	Load Doubleword Indexed X-form
011111 .... 11001 10101/	III	..XX			lhzcix	v2.05	HV		1097	Load Halfword and Zero Caching Inhibited Indexed X-form
111000	I	..XX			lq	v2.03			63	Load Quadword DQ-form
011111 .... 01000 10100.	I	..XX			lqarx	v2.07			1023	Load Quadword And Reserve Indexed X-form
011111 .... 00000 00111/	I	..XX			lvebx	v2.03			260	Load Vector Element Byte Indexed X-form
011111 .... 00001 00111/	I	..XX			lvehx	v2.03			260	Load Vector Element Halfword Indexed X-form
011111 .... 00010 00111/	I	..XX			lvewx	v2.03			261	Load Vector Element Word Indexed X-form
011111 .... 00000 00110/	I	..XX			lvsl	v2.03			267	Load Vector for Shift Left Indexed X-form
011111 .... 00001 00110/	I	..XX			lvsr	v2.03			267	Load Vector for Shift Right Indexed X-form
011111 .... 00011 00111/	I	..XX			lvx	v2.03			262	Load Vector Indexed X-form
011111 .... 01011 00111/	I	..XX			lvxl	v2.03			262	Load Vector Indexed Last X-form
111010 10	I	..XX			lwa	PPC			54	Load Word Algebraic DS-form
011111 .... 01011 10101/	I	..XX			lwaux	PPC			54	Load Word Algebraic with Update Indexed X-form
011111 .... 01010 10101/	I	..XX			lwax	PPC			54	Load Word Algebraic Indexed X-form
011111 .... 11000 10101/	III	..XX			lwzcix	v2.05	HV		1097	Load Word and Zero Caching Inhibited Indexed X-form
111001 10	I	..XX			lxsd	v3.0			562	Load VSX Scalar Doubleword DS-form
011111 .... 10010 01100.	I	..XX			lxsdx	v2.06			563	Load VSX Scalar Doubleword Indexed X-form

Table G.1: Power ISA Instruction Set Sorted by Compliancy Subset (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
011111 .... 11000 01101.	I	..XX			lxsibzx	v3.0			564	Load VSX Scalar as Integer Byte & Zero Indexed X-form
011111 .... 11001 01101.	I	..XX			lxsihzx	v3.0			564	Load VSX Scalar as Integer Halfword & Zero Indexed X-form
011111 .... 00010 01100.	I	..XX			lxsiwax	v2.07			565	Load VSX Scalar as Integer Word Algebraic Indexed X-form
011111 .... 00000 01100.	I	..XX			lxsiwzx	v2.07			566	Load VSX Scalar as Integer Word & Zero Indexed X-form
111001 11	I	..XX			lxssp	v3.0			567	Load VSX Scalar Single-Precision DS-form
011111 .... 10000 01100.	I	..XX			lxsspx	v2.07			568	Load VSX Scalar Single-Precision Indexed X-form
111101 001	I	..XX			lxv	v3.0			575	Load VSX Vector DQ-form
011111 .... 11011 01100.	I	..XX			lxvb16x	v3.0			576	Load VSX Vector Byte*16 Indexed X-form
011111 .... 11010 01100.	I	..XX			lxvd2x	v2.06			577	Load VSX Vector Doubleword*2 Indexed X-form
011111 .... 01010 01100.	I	..XX			lxvdsx	v2.06			582	Load VSX Vector Doubleword & Splat Indexed X-form
011111 .... 11001 01100.	I	..XX			lxvh8x	v3.0			578	Load VSX Vector Halfword*8 Indexed X-form
111100 ..... 11111 ..... 01011 01000.	I	..XX			lxvkq	v3.1			935	Load VSX Vector Special Value Quadword X-form
011111 .... 01000 01101.	I	..XX			lxvl	v3.0			588	Load VSX Vector with Length X-form
011111 .... 01001 01101.	I	..XX			lxvll	v3.0			590	Load VSX Vector with Length Left-justified X-form
000110 0000	I	..XX			lxvp	v3.1			603	Load VSX Vector Paired DQ-form
011111 .... 01010 01101/	I	..XX			lxvpx	v3.1			604	Load VSX Vector Paired Indexed X-form
011111 .... 00000 01101.	I	..XX			lxvrbx	v3.1			584	Load VSX Vector Rightmost Byte Indexed X-form
011111 .... 00011 01101.	I	..XX			lxvrdx	v3.1			585	Load VSX Vector Rightmost Doubleword Indexed X-form
011111 .... 00001 01101.	I	..XX			lxvrhx	v3.1			586	Load VSX Vector Rightmost Halfword Indexed X-form
011111 .... 00010 01101.	I	..XX			lxvrwx	v3.1			587	Load VSX Vector Rightmost Word Indexed X-form
011111 .... 11000 01100.	I	..XX			lxvw4x	v2.06			579	Load VSX Vector Word*4 Indexed X-form
011111 .... 01011 01100.	I	..XX			lxvwsx	v3.0			583	Load VSX Vector Word & Splat Indexed X-form
011111 .... 0100/ 01100.	I	..XX			lxvx	v3.0			580	Load VSX Vector Indexed X-form
000100 110000	I	..XX			maddhd	v3.0			86	Multiply-Add High Doubleword VA-form
000100 110001	I	..XX			maddhdu	v3.0			86	Multiply-Add High Doubleword Unsigned VA-form
000100 110011	I	..XX			maddld	v3.0			86	Multiply-Add Low Doubleword VA-form
000100 ..... ///// ///// 11000 000100	I	..XX			mfvscr	v2.03			483	Move From Vector Status and Control Register VX-form
011111 .... ///// 00001 10011.	I	..XX			mfvsrd	v2.07			122	Move From VSR Doubleword X-form
011111 .... ///// 01001 10011.	I	..XX			mfvsrld	v3.0			122	Move From VSR Lower Doubleword X-form
011111 .... ///// 00011 10011.	I	..XX			mfvsrwz	v2.07			123	Move From VSR Word and Zero X-form
011111 .... 11000 01001/	I	..XX			modsd	v3.0			89	Modulo Signed Doubleword X-form
011111 .... 01000 01001/	I	..XX			modud	v3.0			89	Modulo Unsigned Doubleword X-form
011111 ///// ///// 00111 01110/	III	..XX			msgclr	v2.07	HV		1275	Message Clear X-form
011111 ///// ///// 00101 01110/	III	..XX			msgclrp	v2.07	P		1276	Message Clear Privileged X-form
011111 ///// ///// 00110 01110/	III	..XX			msgsnd	v2.07	HV		1275	Message Send X-form
011111 ///// ///// 00100 01110/	III	..XX			msgsndp	v2.07	P		1276	Message Send Privileged X-form
011111 ///// ///// ///// 11011 10110/	III	..XX			msgsync	v3.0	HV		1277	Message Synchronize X-form
011111 ..... ////. ///// 00101 10010/	III	..XX			mtmsrd	PPC	P		1109	Move To Machine State Register Doubleword X-form
000100 ///// ///// 11001 000100	I	..XX			mtvscr	v2.03			483	Move To Vector Status and Control Register VX-form
000100 ..... 10000 ..... 11001 000010	I	..XX			mtvsrbm	v3.1			451	Move to VSR Byte Mask VX-form
000100 .... 01010.	I	..XX			mtvsrbmi	v3.1			453	Move To VSR Byte Mask Immediate DX-form
011111 .... ///// 00101 10011.	I	..XX			mtvsrd	v2.07			123	Move To VSR Doubleword X-form
011111 .... 01101 10011.	I	..XX			mtvsrdd	v3.0			125	Move To VSR Double Doubleword X-form
000100 ..... 10011 ..... 11001 000010	I	..XX			mtvsrdm	v3.1			452	Move to VSR Doubleword Mask VX-form
000100 ..... 10001 ..... 11001 000010	I	..XX			mtvsrhm	v3.1			451	Move to VSR Halfword Mask VX-form
000100 ..... 10100 ..... 11001 000010	I	..XX			mtvsrqm	v3.1			453	Move to VSR Quadword Mask VX-form

Table G.1: Power ISA Instruction Set Sorted by Compliancy Subset (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
011111 .... ///// 00110 10011.	I	..XX			mtvsrwa	v2.07			124	Move To VSR Word Algebraic X-form
000100 ..... 10010 ..... 11001 000010	I	..XX			mtvsrwm	v3.1			452	Move to VSR Word Mask VX-form
011111 .... ///// 01100 10011.	I	..XX			mtvsrws	v3.0			125	Move To VSR Word & Splat X-form
011111 .... ///// 00111 10011.	I	..XX			mtvsrwz	v2.07			124	Move To VSR Word and Zero X-form
011111 .... /0010 01001.	I	..XX			mulhd[.]	PPC		SR	85	Multiply High Doubleword XO-form
011111 .... /0000 01001.	I	..XX			mulhdu[.]	PPC		SR	85	Multiply High Doubleword Unsigned XO-form
011111 .... 00111 01001.	I	..XX			mulld[.]	PPC		SR	85	Multiply Low Doubleword XO-form
011111 .... 10111 01001.	I	..XX			mulldo[.]	PPC		SR	85	Multiply Low Doubleword & record OV XO-form
011111 .... 10111 01011.	I	..XX			mullwo[.]	P1		SR	79	Multiply Low Word & record OV XO-form
011111 .... ///// 10011 01000.	I	..XX			nego[.]	P1		SR	78	Negate & record OV XO-form
000001 100// .//.. .... 001110	I	..XX			paddi	v3.1			73	Prefixed Add Immediate MLS:D-form
011111 .... 00100 11100/	I	..XX			pdepd	v3.1			106	Parallel Bits Deposit Doubleword X-form
011111 .... 00101 11100/	I	..XX			pextd	v3.1			106	Parallel Bits Extract Doubleword X-form
000001 100// .//.. .... 100010	I	..XX			plbz	v3.1			48	Prefixed Load Byte and Zero MLS:D-form
000001 000// .//.. .... 111001	I	..XX			pld	v3.1			55	Prefixed Load Doubleword 8LS:D-form
000001 100// .//.. .... 110010	I	..XX			plfd	v3.1			152	Prefixed Load Floating-Point Double MLS:D-form
000001 100// .//.. .... 110000	I	..XX			plfs	v3.1			150	Prefixed Load Floating-Point Single MLS:D-form
000001 100// .//.. .... 101010	I	..XX			plha	v3.1			51	Prefixed Load Halfword Algebraic MLS:D-form
000001 100// .//.. .... 101000	I	..XX			plhz	v3.1			50	Prefixed Load Halfword and Zero MLS:D-form
000001 000// .//.. .... 111000	I	..XX			plq	v3.1			63	Prefixed Load Quadword 8LS:D-form
000001 000// .//.. .... 101001	I	..XX			plwa	v3.1			54	Prefixed Load Word Algebraic 8LS:D-form
000001 100// .//.. .... 100000	I	..XX			plwz	v3.1			53	Prefixed Load Word and Zero MLS:D-form
000001 000// .//.. .... 101010	I	..XX			plxsd	v3.1			562	Prefixed Load VSX Scalar Doubleword 8LS:D-form
000001 000// .//.. .... 101011	I	..XX			plxssp	v3.1			567	Prefixed Load VSX Scalar Single-Precision 8LS:D-form
000001 000// .//.. .... 11001.	I	..XX			plxv	v3.1			575	Prefixed Load VSX Vector 8LS:D-form
000001 000// .//.. .... 111010	I	..XX			plxvp	v3.1			603	Prefixed Load VSX Vector Paired 8LS:D-form
000001 11000 0//00 00000 00000 000000 ?????? ????? ????? ????? ????? ??????	I	..XX			pnop	v3.1			132	Prefixed Nop MRR:*-form
011111 .... ///// 01111 11010/	I	..XX			popcntd	v2.06			103	Population Count Doubleword X-form
011111 .... ///// 00101 11010/	I	..XX			prtyd	v2.05			103	Parity Doubleword X-form
000001 100// .//.. .... 100110	I	..XX			pstb	v3.1			57	Prefixed Store Byte MLS:D-form
000001 000// .//.. .... 111101	I	..XX			pstd	v3.1			60	Prefixed Store Doubleword 8LS:D-form

Table G.1: Power ISA Instruction Set Sorted by Compliancy Subset (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
000001 100// .//.. .... 110110	I	..XX			pstfd	v3.1			156	Prefixed Store Floating-Point Double MLS:D-form
000001 100// .//.. .... 110100	I	..XX			pstfs	v3.1			154	Prefixed Store Floating-Point Single MLS:D-form
000001 100// .//.. .... 101100	I	..XX			psth	v3.1			58	Prefixed Store Halfword MLS:D-form
000001 000// .//.. .... 111100	I	..XX			pstq	v3.1			64	Prefixed Store Quadword 8LS:D-form
000001 100// .//.. .... 100100	I	..XX			pstw	v3.1			59	Prefixed Store Word MLS:D-form
000001 000// .//.. .... 101110	I	..XX			pstxsd	v3.1			569	Prefixed Store VSX Scalar Doubleword 8LS:D-form
000001 000// .//.. .... 101111	I	..XX			pstxssp	v3.1			573	Prefixed Store VSX Scalar Single-Precision 8LS:D-form
000001 000// .//.. .... 11011.	I	..XX			pstxv	v3.1			591	Prefixed Store VSX Vector 8LS:D-form
000001 000// .//.. .... 111110	I	..XX			pstxvp	v3.1			605	Prefixed Store VSX Vector Paired 8LS:D-form
011110 .... .1000.	I	..XX			rldcl[.]	PPC		SR	111	Rotate Left Doubleword then Clear Left MDS-form
011110 .... .1001.	I	..XX			rldcr[.]	PPC		SR	112	Rotate Left Doubleword then Clear Right MDS-form
011110 .... .010..	I	..XX			rldic[.]	PPC		SR	111	Rotate Left Doubleword Immediate then Clear MD-form
011110 .... .000..	I	..XX			rldicl[.]	PPC		SR	110	Rotate Left Doubleword Immediate then Clear Left MD-form
011110 .... .001..	I	..XX			rldicr[.]	PPC		SR	110	Rotate Left Doubleword Immediate then Clear Right MD-form
011110 .... .011..	I	..XX			rldimi[.]	PPC		SR	112	Rotate Left Doubleword Immediate then Mask Insert MD-form
011111 //... ///// ///// 01111 10010/	III	..XX			slbia	PPC	P		1164	SLB Invalidate All X-form
011111 .... 00000 11011.	I	..XX			sld[.]	PPC		SR	115	Shift Left Doubleword X-form
011111 .... 11000 11010.	I	..XX			srad[.]	PPC		SR	115	Shift Right Algebraic Doubleword X-form
011111 .... 11001 1101..	I	..XX			sradi[.]	PPC		SR	115	Shift Right Algebraic Doubleword Immediate XS-form
011111 .... 10000 11011.	I	..XX			srd[.]	PPC		SR	115	Shift Right Doubleword X-form
011111 .... 11110 10101/	III	..XX			stbcix	v2.05	HV		1098	Store Byte Caching Inhibited Indexed X-form
111110 00	I	..XX			std	PPC			60	Store Doubleword DS-form
011111 .... 10100 10100/	I	..XX			stdbrx	v2.06			67	Store Doubleword Byte-Reverse Indexed X-form
011111 .... 11111 10101/	III	..XX			stdcix	v2.05	HV		1098	Store Doubleword Caching Inhibited Indexed X-form
011111 .... 00110 101101	II	..XX			stdcx.	PPC			1022	Store Doubleword Conditional Indexed X-form
111110 01	I	..XX			stdu	PPC			61	Store Doubleword with Update DS-form
011111 .... 00101 10101/	I	..XX			stdux	PPC			61	Store Doubleword with Update Indexed X-form
011111 .... 00100 10101/	I	..XX			stdx	PPC			60	Store Doubleword Indexed X-form
011111 .... 11101 10101/	III	..XX			sthcix	v2.05	HV		1098	Store Halfword Caching Inhibited Indexed X-form
010011 ///// ///// ///// 01011 10010/	III	..XX			stop	v3.0	P		1089	Stop XL-form
111110 10	I	..XX			stq	v2.03			64	Store Quadword DS-form
011111 .... 00101 101101	I	..XX			stqcx.	v2.07			1024	Store Quadword Conditional Indexed X-form
011111 .... 00100 00111/	I	..XX			stvebx	v2.03			263	Store Vector Element Byte Indexed X-form
011111 .... 00101 00111/	I	..XX			stvehx	v2.03			264	Store Vector Element Halfword Indexed X-form
011111 .... 00110 00111/	I	..XX			stvewx	v2.03			264	Store Vector Element Word Indexed X-form
011111 .... 00111 00111/	I	..XX			stvx	v2.03			265	Store Vector Indexed X-form
011111 .... 01111 00111/	I	..XX			stvxl	v2.03			265	Store Vector Indexed Last X-form

Table G.1: Power ISA Instruction Set Sorted by Compliancy Subset (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
011111 .... 11100 10101/	III	..XX			stwcix	v2.05	HV		1098	Store Word Caching Inhibited Indexed X-form
111101 10	I	..XX			stxsd	v3.0			569	Store VSX Scalar Doubleword DS-form
011111 .... 10110 01100.	I	..XX			stxsdx	v2.06			570	Store VSX Scalar Doubleword Indexed X-form
011111 .... 11100 01101.	I	..XX			stxsibx	v3.0			571	Store VSX Scalar as Integer Byte Indexed X-form
011111 .... 11101 01101.	I	..XX			stxsihx	v3.0			571	Store VSX Scalar as Integer Halfword Indexed X-form
011111 .... 00100 01100.	I	..XX			stxsiwx	v2.07			572	Store VSX Scalar as Integer Word Indexed X-form
111101 11	I	..XX			stxssp	v3.0			573	Store VSX Scalar Single-Precision DS-form
011111 .... 10100 01100.	I	..XX			stxsspx	v2.07			574	Store VSX Scalar Single-Precision Indexed X-form
111101 101	I	..XX			stxv	v3.0			591	Store VSX Vector DQ-form
011111 .... 11111 01100.	I	..XX			stxvb16x	v3.0			592	Store VSX Vector Byte*16 Indexed X-form
011111 .... 11110 01100.	I	..XX			stxvd2x	v2.06			593	Store VSX Vector Doubleword*2 Indexed X-form
011111 .... 11101 01100.	I	..XX			stxvh8x	v3.0			594	Store VSX Vector Halfword*8 Indexed X-form
011111 .... 01100 01101.	I	..XX			stxvl	v3.0			600	Store VSX Vector with Length X-form
011111 .... 01101 01101.	I	..XX			stxvll	v3.0			602	Store VSX Vector with Length Left-justified X-form
000110 0001	I	..XX			stxvp	v3.1			605	Store VSX Vector Paired DQ-form
011111 .... 01110 01101/	I	..XX			stxvpx	v3.1			606	Store VSX Vector Paired Indexed X-form
011111 .... 00100 01101.	I	..XX			stxvrbx	v3.1			598	Store VSX Vector Rightmost Byte Indexed X-form
011111 .... 00111 01101.	I	..XX			stxvrdx	v3.1			598	Store VSX Vector Rightmost Doubleword Indexed X-form
011111 .... 00101 01101.	I	..XX			stxvrhx	v3.1			599	Store VSX Vector Rightmost Halfword Indexed X-form
011111 .... 00110 01101.	I	..XX			stxvrwx	v3.1			599	Store VSX Vector Rightmost Word Indexed X-form
011111 .... 11100 01100.	I	..XX			stxvw4x	v2.06			595	Store VSX Vector Word*4 Indexed X-form
011111 .... 01100 01100.	I	..XX			stxvx	v3.0			596	Store VSX Vector Indexed X-form
011111 .... 10000 01000.	I	..XX			subfco[.]	P1		SR	76	Subtract From Carrying & record OV XO-form
011111 .... 10100 01000.	I	..XX			subfeo[.]	P1		SR	76	Subtract From Extended & record OV XO-form
011111 .... ///// 10111 01000.	I	..XX			subfmeo[.]	P1		SR	77	Subtract From Minus One Extended & record OV XO-form
011111 .... 10001 01000.	I	..XX			subfo[.]	PPC		SR	75	Subtract From & record OV XO-form
011111 .... ///// 10110 01000.	I	..XX			subfzeo[.]	P1		SR	77	Subtract From Zero Extended & record OV XO-form
011111 .... 00010 00100/	I	..XX			td	PPC			95	Trap Doubleword X-form
000010	I	..XX			tdi	PPC			95	Trap Doubleword Immediate D-form
011111 ..... /.... ..... 01000 10010/	III	..XX			tlbiel	v2.03	P	64	1181	TLB Invalidate Entry Local X-form
000100 .... 10000 000011	I	..XX			vabsdub	v3.0			368	Vector Absolute Difference Unsigned Byte VX-form
000100 .... 10001 000011	I	..XX			vabsduh	v3.0			368	Vector Absolute Difference Unsigned Halfword VX-form
000100 .... 10010 000011	I	..XX			vabsduw	v3.0			369	Vector Absolute Difference Unsigned Word VX-form
000100 .... 00101 000000	I	..XX			vaddcuq	v2.07			320	Vector Add & write Carry-out Unsigned Quadword VX-form
000100 .... 00110 000000	I	..XX			vaddcuw	v2.03			313	Vector Add & write Carry-out Unsigned Word VX-form
000100 111101	I	..XX			vaddecuq	v2.07			320	Vector Add Extended & write Carry-out Unsigned Quadword VA-form
000100 111100	I	..XX			vaddeuqm	v2.07			319	Vector Add Extended Unsigned Quadword Modulo VA-form
000100 .... 00000 001010	I	..XX			vaddfp	v2.03			411	Vector Add Floating-Point VX-form
000100 .... 01100 000000	I	..XX			vaddsbs	v2.03			313	Vector Add Signed Byte Saturate VX-form
000100 .... 01101 000000	I	..XX			vaddshs	v2.03			314	Vector Add Signed Halfword Saturate VX-form
000100 .... 01110 000000	I	..XX			vaddsws	v2.03			314	Vector Add Signed Word Saturate VX-form
000100 .... 00000 000000	I	..XX			vaddubm	v2.03			315	Vector Add Unsigned Byte Modulo VX-form
000100 .... 01000 000000	I	..XX			vaddubs	v2.03			317	Vector Add Unsigned Byte Saturate VX-form
000100 .... 00011 000000	I	..XX			vaddudm	v2.07			316	Vector Add Unsigned Doubleword Modulo VX-form
000100 .... 00001 000000	I	..XX			vadduhm	v2.03			315	Vector Add Unsigned Halfword Modulo VX-form

Table G.1: Power ISA Instruction Set Sorted by Compliancy Subset (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
000100 .... 01001 000000	I	..XX			vadduhs	v2.03			317	Vector Add Unsigned Halfword Saturate VX-form
000100 .... 00100 000000	I	..XX			vadduqm	v2.07			319	Vector Add Unsigned Quadword Modulo VX-form
000100 .... 00010 000000	I	..XX			vadduwm	v2.03			316	Vector Add Unsigned Word Modulo VX-form
000100 .... 01010 000000	I	..XX			vadduws	v2.03			318	Vector Add Unsigned Word Saturate VX-form
000100 .... 10000 000100	I	..XX			vand	v2.03			392	Vector Logical AND VX-form
000100 .... 10001 000100	I	..XX			vandc	v2.03			392	Vector Logical AND with Complement VX-form
000100 .... 10100 000010	I	..XX			vavgsb	v2.03			365	Vector Average Signed Byte VX-form
000100 .... 10101 000010	I	..XX			vavgsh	v2.03			366	Vector Average Signed Halfword VX-form
000100 .... 10110 000010	I	..XX			vavgsw	v2.03			367	Vector Average Signed Word VX-form
000100 .... 10000 000010	I	..XX			vavgub	v2.03			365	Vector Average Unsigned Byte VX-form
000100 .... 10001 000010	I	..XX			vavguh	v2.03			366	Vector Average Unsigned Halfword VX-form
000100 .... 10010 000010	I	..XX			vavguw	v2.03			367	Vector Average Unsigned Word VX-form
000100 .... 10111 001100	I	..XX			vbpermd	v3.0			449	Vector Bit Permute Doubleword VX-form
000100 .... 10101 001100	I	..XX			vbpermq	v2.07			450	Vector Bit Permute Quadword VX-form
000100 .... 01101 001010	I	..XX			vcfsx	v2.03			415	Vector Convert with round to nearest From Signed Word to floating-point format VX-form
000100 .... 10101 001101	I	..XX			vcfuged	v3.1			444	Vector Centrifuge Doubleword VX-form
000100 .... 01100 001010	I	..XX			vcfux	v2.03			415	Vector Convert with round to nearest From Unsigned Word to floating-point format VX-form
000100 .... 10100 001000	I	..XX			vcipher	v2.07			424	Vector AES Cipher VX-form
000100 .... 10100 001001	I	..XX			vcipherlast	v2.07			424	Vector AES Cipher Last VX-form
000100 .... 00110 001101	I	..XX			vclrlb	v3.1			464	Vector Clear Leftmost Bytes VX-form
000100 .... 00111 001101	I	..XX			vclrrb	v3.1			464	Vector Clear Rightmost Bytes VX-form
000100 ..... ///// 11100 000010	I	..XX			vclzb	v2.07			435	Vector Count Leading Zeros Byte VX-form
000100 ..... ///// 11111 000010	I	..XX			vclzd	v2.07			437	Vector Count Leading Zeros Doubleword VX-form
000100 .... 11110 000100	I	..XX			vclzdm	v3.1			437	Vector Count Leading Zeros Doubleword under bit Mask VX-form
000100 ..... ///// 11101 000010	I	..XX			vclzh	v2.07			435	Vector Count Leading Zeros Halfword VX-form
000100 ..... 00000 ..... 11000 000010	I	..XX			vclzlsbb	v3.0			441	Vector Count Leading Zero Least-Significant Bits Byte VX-form
000100 ..... ///// 11110 000010	I	..XX			vclzw	v2.07			436	Vector Count Leading Zeros Word VX-form
000100 .... .1111 000110	I	..XX			vcmpbfp[.]	v2.03			418	Vector Compare Bounds Floating-Point VC-form
000100 .... .0011 000110	I	..XX			vcmpeqfp[.]	v2.03			419	Vector Compare Equal Floating-Point VC-form
000100 .... .0000 000110	I	..XX			vcmpequb[.]	v2.03			378	Vector Compare Equal Unsigned Byte VC-form
000100 .... .0011 000111	I	..XX			vcmpequd[.]	v2.07			381	Vector Compare Equal Unsigned Doubleword VC-form
000100 .... .0001 000110	I	..XX			vcmpequh[.]	v2.03			379	Vector Compare Equal Unsigned Halfword VC-form
000100 .... .0111 000111	I	..XX			vcmpequq[.]	v3.1			382	Vector Compare Equal Quadword VC-form
000100 .... .0010 000110	I	..XX			vcmpequw[.]	v2.03			380	Vector Compare Equal Unsigned Word VC-form
000100 .... .0111 000110	I	..XX			vcmpgefp[.]	v2.03			419	Vector Compare Greater Than or Equal Floating-Point VC-form
000100 .... .1011 000110	I	..XX			vcmpgtfp[.]	v2.03			420	Vector Compare Greater Than Floating-Point VC-form
000100 .... .1100 000110	I	..XX			vcmpgtsb[.]	v2.03			383	Vector Compare Greater Than Signed Byte VC-form
000100 .... .1111 000111	I	..XX			vcmpgtsd[.]	v2.07			386	Vector Compare Greater Than Signed Doubleword VC-form
000100 .... .1101 000110	I	..XX			vcmpgtsh[.]	v2.03			384	Vector Compare Greater Than Signed Halfword VC-form
000100 .... .1110 000111	I	..XX			vcmpgtsq[.]	v3.1			387	Vector Compare Greater Than Signed Quadword VC-form
000100 .... .1110 000110	I	..XX			vcmpgtsw[.]	v2.03			385	Vector Compare Greater Than Signed Word VC-form
000100 .... .1000 000110	I	..XX			vcmpgtub[.]	v2.03			383	Vector Compare Greater Than Unsigned Byte VC-form

Table G.1: Power ISA Instruction Set Sorted by Compliancy Subset (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
000100 .... .1011 000111	I	..XX			vcmpgtud[.]	v2.07			386	Vector Compare Greater Than Unsigned Doubleword VC-form
000100 .... .1001 000110	I	..XX			vcmpgtuh[.]	v2.03			384	Vector Compare Greater Than Unsigned Halfword VC-form
000100 .... .1010 000111	I	..XX			vcmpgtuq[.]	v3.1			387	Vector Compare Greater Than Unsigned Quadword VC-form
000100 .... .1010 000110	I	..XX			vcmpgtuw[.]	v2.03			385	Vector Compare Greater Than Unsigned Word VC-form
000100 .... .0000 000111	I	..XX			vcmpneb[.]	v3.0			388	Vector Compare Not Equal Byte VC-form
000100 .... .0001 000111	I	..XX			vcmpneh[.]	v3.0			389	Vector Compare Not Equal Halfword VC-form
000100 .... .0010 000111	I	..XX			vcmpnew[.]	v3.0			390	Vector Compare Not Equal Word VC-form
000100 .... .0100 000111	I	..XX			vcmpnezb[.]	v3.0			388	Vector Compare Not Equal or Zero Byte VC-form
000100 .... .0101 000111	I	..XX			vcmpnezh[.]	v3.0			389	Vector Compare Not Equal or Zero Halfword VC-form
000100 .... .0110 000111	I	..XX			vcmpnezw[.]	v3.0			390	Vector Compare Not Equal or Zero Word VC-form
000100 .../// 00101 000001	I	..XX			vcmpsq	v3.1			391	Vector Compare Signed Quadword VX-form
000100 .../// 00100 000001	I	..XX			vcmpuq	v3.1			391	Vector Compare Unsigned Quadword VX-form
000100 ..... 1100. ..... 11001 000010	I	..XX			vcntmbb	v3.1			457	Vector Count Mask Bits Byte VX-form
000100 ..... 1111. ..... 11001 000010	I	..XX			vcntmbd	v3.1			458	Vector Count Mask Bits Doubleword VX-form
000100 ..... 1101. ..... 11001 000010	I	..XX			vcntmbh	v3.1			457	Vector Count Mask Bits Halfword VX-form
000100 ..... 1110. ..... 11001 000010	I	..XX			vcntmbw	v3.1			458	Vector Count Mask Bits Word VX-form
000100 .... 01111 001010	I	..XX			vctsxs	v2.03			414	Vector Convert with round to zero from floating-point To Signed Word format Saturate VX-form
000100 .... 01110 001010	I	..XX			vctuxs	v2.03			414	Vector Convert with round to zero from floating-point To Unsigned Word format Saturate VX-form
000100 ..... 11100 ..... 11000 000010	I	..XX			vctzb	v3.0			438	Vector Count Trailing Zeros Byte VX-form
000100 ..... 11111 ..... 11000 000010	I	..XX			vctzd	v3.0			440	Vector Count Trailing Zeros Doubleword VX-form
000100 .... 11111 000100	I	..XX			vctzdm	v3.1			440	Vector Count Trailing Zeros Doubleword under bit Mask VX-form
000100 ..... 11101 ..... 11000 000010	I	..XX			vctzh	v3.0			438	Vector Count Trailing Zeros Halfword VX-form
000100 ..... 00001 ..... 11000 000010	I	..XX			vctzlsbb	v3.0			441	Vector Count Trailing Zero Least-Significant Bits Byte VX-form
000100 ..... 11110 ..... 11000 000010	I	..XX			vctzw	v3.0			439	Vector Count Trailing Zeros Word VX-form
000100 .... 01111 001011	I	..XX			vdivesd	v3.1			351	Vector Divide Extended Signed Doubleword VX-form
000100 .... 01100 001011	I	..XX			vdivesq	v3.1			353	Vector Divide Extended Signed Quadword VX-form
000100 .... 01110 001011	I	..XX			vdivesw	v3.1			349	Vector Divide Extended Signed Word VX-form
000100 .... 01011 001011	I	..XX			vdiveud	v3.1			351	Vector Divide Extended Unsigned Doubleword VX-form
000100 .... 01000 001011	I	..XX			vdiveuq	v3.1			353	Vector Divide Extended Unsigned Quadword VX-form
000100 .... 01010 001011	I	..XX			vdiveuw	v3.1			349	Vector Divide Extended Unsigned Word VX-form
000100 .... 00111 001011	I	..XX			vdivsd	v3.1			350	Vector Divide Signed Doubleword VX-form
000100 .... 00100 001011	I	..XX			vdivsq	v3.1			352	Vector Divide Signed Quadword VX-form
000100 .... 00110 001011	I	..XX			vdivsw	v3.1			348	Vector Divide Signed Word VX-form
000100 .... 00011 001011	I	..XX			vdivud	v3.1			350	Vector Divide Unsigned Doubleword VX-form
000100 .... 00000 001011	I	..XX			vdivuq	v3.1			352	Vector Divide Unsigned Quadword VX-form
000100 .... 00010 001011	I	..XX			vdivuw	v3.1			348	Vector Divide Unsigned Word VX-form
000100 .... 11010 000100	I	..XX			veqv	v2.07			393	Vector Logical Equivalence VX-form
000100 ..... 00000 ..... 11001 000010	I	..XX			vexpandbm	v3.1			454	Vector Expand Byte Mask VX-form
000100 ..... 00011 ..... 11001 000010	I	..XX			vexpanddm	v3.1			455	Vector Expand Doubleword Mask VX-form
000100 ..... 00001 ..... 11001 000010	I	..XX			vexpandhm	v3.1			454	Vector Expand Halfword Mask VX-form
000100 ..... 00100 ..... 11001 000010	I	..XX			vexpandqm	v3.1			456	Vector Expand Quadword Mask VX-form

Table G.1: Power ISA Instruction Set Sorted by Compliancy Subset (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
000100 ..... 00010 ..... 11001 000010	I	..XX			vexpandwm	v3.1			455	Vector Expand Word Mask VX-form
000100 ..... ///// 00110 001010	I	..XX			vexptefp	v2.03			421	Vector 2 Raised to the Exponent Estimate Floating-Point VX-form
000100 011110	I	..XX			vextddvlx	v3.1			302	Vector Extract Double Doubleword to VSR using GPR-specified Left-Index VA-form
000100 011111	I	..XX			vextddvrx	v3.1			302	Vector Extract Double Doubleword to VSR using GPR-specified Right-Index VA-form
000100 011000	I	..XX			vextdubvlx	v3.1			299	Vector Extract Double Unsigned Byte to VSR using GPR-specified Left-Index VA-form
000100 011001	I	..XX			vextdubvrx	v3.1			299	Vector Extract Double Unsigned Byte to VSR using GPR-specified Right-Index VA-form
000100 011010	I	..XX			vextduhvlx	v3.1			300	Vector Extract Double Unsigned Halfword to VSR using GPR-specified Left-Index VA-form
000100 011011	I	..XX			vextduhvrx	v3.1			300	Vector Extract Double Unsigned Halfword to VSR using GPR-specified Right-Index VA-form
000100 011100	I	..XX			vextduwvlx	v3.1			301	Vector Extract Double Unsigned Word to VSR using GPR-specified Left-Index VA-form
000100 011101	I	..XX			vextduwvrx	v3.1			301	Vector Extract Double Unsigned Word to VSR using GPR-specified Right-Index VA-form
000100 ..... 01000 ..... 11001 000010	I	..XX			vextractbm	v3.1			459	Vector Extract Byte Mask VX-form
000100 ..... /.... ..... 01011 001101	I	..XX			vextractd	v3.0			295	Vector Extract Doubleword to VSR using immediate-specified index VX-form
000100 ..... 01011 ..... 11001 000010	I	..XX			vextractdm	v3.1			460	Vector Extract Doubleword Mask VX-form
000100 ..... 01001 ..... 11001 000010	I	..XX			vextracthm	v3.1			459	Vector Extract Halfword Mask VX-form
000100 ..... 01100 ..... 11001 000010	I	..XX			vextractqm	v3.1			461	Vector Extract Quadword Mask VX-form
000100 ..... /.... ..... 01000 001101	I	..XX			vextractub	v3.0			294	Vector Extract Unsigned Byte to VSR using immediate-specified index VX-form
000100 ..... /.... ..... 01001 001101	I	..XX			vextractuh	v3.0			294	Vector Extract Unsigned Halfword to VSR using immediate-specified index VX-form
000100 ..... /.... ..... 01010 001101	I	..XX			vextractuw	v3.0			295	Vector Extract Unsigned Word to VSR using immediate-specified index VX-form
000100 ..... 01010 ..... 11001 000010	I	..XX			vextractwm	v3.1			460	Vector Extract Word Mask VX-form
000100 ..... 11000 ..... 11000 000010	I	..XX			vextsb2d	v3.0			363	Vector Extend Sign Byte To Doubleword VX-form
000100 ..... 10000 ..... 11000 000010	I	..XX			vextsb2w	v3.0			362	Vector Extend Sign Byte To Word VX-form
000100 ..... 11011 ..... 11000 000010	I	..XX			vextsd2q	v3.1			364	Vector Extend Sign Doubleword to Quadword VX-form
000100 ..... 11001 ..... 11000 000010	I	..XX			vextsh2d	v3.0			363	Vector Extend Sign Halfword To Doubleword VX-form
000100 ..... 10001 ..... 11000 000010	I	..XX			vextsh2w	v3.0			362	Vector Extend Sign Halfword To Word VX-form
000100 ..... 11010 ..... 11000 000010	I	..XX			vextsw2d	v3.0			364	Vector Extend Sign Word To Doubleword VX-form
000100 .... 11000 001101	I	..XX			vextublx	v3.0			296	Vector Extract Unsigned Byte to GPR using GPR-specified Left-Index VX-form
000100 .... 11100 001101	I	..XX			vextubrx	v3.0			296	Vector Extract Unsigned Byte to GPR using GPR-specified Right-Index VX-form
000100 .... 11001 001101	I	..XX			vextuhlx	v3.0			297	Vector Extract Unsigned Halfword to GPR using GPR-specified Left-Index VX-form
000100 .... 11101 001101	I	..XX			vextuhrx	v3.0			297	Vector Extract Unsigned Halfword to GPR using GPR-specified Right-Index VX-form
000100 .... 11010 001101	I	..XX			vextuwlx	v3.0			298	Vector Extract Unsigned Word to GPR using GPR-specified Left-Index VX-form

Table G.1: Power ISA Instruction Set Sorted by Compliancy Subset (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
000100 .... 11110 001101	I	..XX			vextuwrx	v3.0			298	Vector Extract Unsigned Word to GPR using GPR-specified Right-Index VX-form
000100 ..... ///// 10100 001100	I	..XX			vgbbd	v2.07			433	Vector Gather Bits by Bytes by Doubleword VX-form
000100 ..... //... ..... 10011 001100	I	..XX			vgnb	v3.1			434	Vector Gather every Nth Bit VX-form
000100 .... 01000 001111	I	..XX			vinsblx	v3.1			305	Vector Insert Byte from GPR using GPR-specified Left-Index VX-form
000100 .... 01100 001111	I	..XX			vinsbrx	v3.1			305	Vector Insert Byte from GPR using GPR-specified Right-Index VX-form
000100 .... 00000 001111	I	..XX			vinsbvlx	v3.1			310	Vector Insert Byte from VSR using GPR-specified Left-Index VX-form
000100 .... 00100 001111	I	..XX			vinsbvrx	v3.1			310	Vector Insert Byte from VSR using GPR-specified Right-Index VX-form
000100 ..... /.... ..... 00111 001111	I	..XX			vinsd	v3.1			309	Vector Insert Doubleword from GPR using immediate-specified index VX-form
000100 .... 01011 001111	I	..XX			vinsdlx	v3.1			308	Vector Insert Doubleword from GPR using GPR-specified Left-Index VX-form
000100 .... 01111 001111	I	..XX			vinsdrx	v3.1			308	Vector Insert Doubleword from GPR using GPR-specified Right-Index VX-form
000100 ..... /.... ..... 01100 001101	I	..XX			vinsertb	v3.0			303	Vector Insert Byte from VSR using immediate-specified index VX-form
000100 ..... /.... ..... 01111 001101	I	..XX			vinsertd	v3.0			304	Vector Insert Doubleword from VSR using immediate-specified index VX-form
000100 ..... /.... ..... 01101 001101	I	..XX			vinserth	v3.0			303	Vector Insert Halfword from VSR using immediate-specified index VX-form
000100 ..... /.... ..... 01110 001101	I	..XX			vinsertw	v3.0			304	Vector Insert Word from VSR using immediate-specified index VX-form
000100 .... 01001 001111	I	..XX			vinshlx	v3.1			306	Vector Insert Halfword from GPR using GPR-specified Left-Index VX-form
000100 .... 01101 001111	I	..XX			vinshrx	v3.1			306	Vector Insert Halfword from GPR using GPR-specified Right-Index VX-form
000100 .... 00001 001111	I	..XX			vinshvlx	v3.1			311	Vector Insert Halfword from VSR using GPR-specified Left-Index VX-form
000100 .... 00101 001111	I	..XX			vinshvrx	v3.1			311	Vector Insert Halfword from VSR using GPR-specified Right-Index VX-form
000100 ..... /.... ..... 00011 001111	I	..XX			vinsw	v3.1			309	Vector Insert Word from GPR using immediate-specified index VX-form
000100 .... 01010 001111	I	..XX			vinswlx	v3.1			307	Vector Insert Word from GPR using GPR-specified Left-Index VX-form
000100 .... 01110 001111	I	..XX			vinswrx	v3.1			307	Vector Insert Word from GPR using GPR-specified Right-Index VX-form
000100 .... 00010 001111	I	..XX			vinswvlx	v3.1			312	Vector Insert Word from VSR using GPR-specified Left-Index VX-form
000100 .... 00110 001111	I	..XX			vinswvrx	v3.1			312	Vector Insert Word from VSR using GPR-specified Right-Index VX-form
000100 ..... ///// 00111 001010	I	..XX			vlogefp	v2.03			422	Vector Log Base 2 Estimate Floating-Point VX-form
000100 101110	I	..XX			vmaddfp	v2.03			412	Vector Multiply-Add Floating-Point VA-form
000100 .... 10000 001010	I	..XX			vmaxfp	v2.03			413	Vector Maximum Floating-Point VX-form
000100 .... 00100 000010	I	..XX			vmaxsb	v2.03			370	Vector Maximum Signed Byte VX-form
000100 .... 00111 000010	I	..XX			vmaxsd	v2.07			373	Vector Maximum Signed Doubleword VX-form

Table G.1: Power ISA Instruction Set Sorted by Compliancy Subset (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
000100 .... 00101 000010	I	..XX			vmaxsh	v2.03			371	Vector Maximum Signed Halfword VX-form
000100 .... 00110 000010	I	..XX			vmaxsw	v2.03			372	Vector Maximum Signed Word VX-form
000100 .... 00000 000010	I	..XX			vmaxub	v2.03			370	Vector Maximum Unsigned Byte VX-form
000100 .... 00011 000010	I	..XX			vmaxud	v2.07			373	Vector Maximum Unsigned Doubleword VX-form
000100 .... 00001 000010	I	..XX			vmaxuh	v2.03			371	Vector Maximum Unsigned Halfword VX-form
000100 .... 00010 000010	I	..XX			vmaxuw	v2.03			372	Vector Maximum Unsigned Word VX-form
000100 100000	I	..XX			vmhaddshs	v2.03			341	Vector Multiply-High-Add Signed Halfword Saturate VA-form
000100 100001	I	..XX			vmhraddshs	v2.03			341	Vector Multiply-High-Round-Add Signed Halfword Saturate VA-form
000100 .... 10001 001010	I	..XX			vminfp	v2.03			413	Vector Minimum Floating-Point VX-form
000100 .... 01100 000010	I	..XX			vminsb	v2.03			374	Vector Minimum Signed Byte VX-form
000100 .... 01111 000010	I	..XX			vminsd	v2.07			377	Vector Minimum Signed Doubleword VX-form
000100 .... 01101 000010	I	..XX			vminsh	v2.03			375	Vector Minimum Signed Halfword VX-form
000100 .... 01110 000010	I	..XX			vminsw	v2.03			376	Vector Minimum Signed Word VX-form
000100 .... 01000 000010	I	..XX			vminub	v2.03			374	Vector Minimum Unsigned Byte VX-form
000100 .... 01011 000010	I	..XX			vminud	v2.07			377	Vector Minimum Unsigned Doubleword VX-form
000100 .... 01001 000010	I	..XX			vminuh	v2.03			375	Vector Minimum Unsigned Halfword VX-form
000100 .... 01010 000010	I	..XX			vminuw	v2.03			376	Vector Minimum Unsigned Word VX-form
000100 100010	I	..XX			vmladduhm	v2.03			342	Vector Multiply-Low-Add Unsigned Halfword Modulo VA-form
000100 .... 11111 001011	I	..XX			vmodsd	v3.1			355	Vector Modulo Signed Doubleword VX-form
000100 .... 11100 001011	I	..XX			vmodsq	v3.1			356	Vector Modulo Signed Quadword VX-form
000100 .... 11110 001011	I	..XX			vmodsw	v3.1			354	Vector Modulo Signed Word VX-form
000100 .... 11011 001011	I	..XX			vmodud	v3.1			355	Vector Modulo Unsigned Doubleword VX-form
000100 .... 11000 001011	I	..XX			vmoduq	v3.1			356	Vector Modulo Unsigned Quadword VX-form
000100 .... 11010 001011	I	..XX			vmoduw	v3.1			354	Vector Modulo Unsigned Word VX-form
000100 .... 11110 001100	I	..XX			vmrgew	v2.07			282	Vector Merge Even Word VX-form
000100 .... 00000 001100	I	..XX			vmrghb	v2.03			279	Vector Merge High Byte VX-form
000100 .... 00001 001100	I	..XX			vmrghh	v2.03			280	Vector Merge High Halfword VX-form
000100 .... 00010 001100	I	..XX			vmrghw	v2.03			281	Vector Merge High Word VX-form
000100 .... 00100 001100	I	..XX			vmrglb	v2.03			279	Vector Merge Low Byte VX-form
000100 .... 00101 001100	I	..XX			vmrglh	v2.03			280	Vector Merge Low Halfword VX-form
000100 .... 00110 001100	I	..XX			vmrglw	v2.03			281	Vector Merge Low Word VX-form
000100 .... 11010 001100	I	..XX			vmrgow	v2.07			282	Vector Merge Odd Word VX-form
000100 010111	I	..XX			vmsumcud	v3.1			347	Vector Multiply-Sum & write Carry-out Unsigned Doubleword VA-form
000100 100101	I	..XX			vmsummbm	v2.03			343	Vector Multiply-Sum Mixed Byte Modulo VA-form
000100 101000	I	..XX			vmsumshm	v2.03			343	Vector Multiply-Sum Signed Halfword Modulo VA-form
000100 101001	I	..XX			vmsumshs	v2.03			344	Vector Multiply-Sum Signed Halfword Saturate VA-form
000100 100100	I	..XX			vmsumubm	v2.03			342	Vector Multiply-Sum Unsigned Byte Modulo VA-form
000100 100011	I	..XX			vmsumudm	v3.0B			346	Vector Multiply-Sum Unsigned Doubleword Modulo VA-form
000100 100110	I	..XX			vmsumuhm	v2.03			344	Vector Multiply-Sum Unsigned Halfword Modulo VA-form
000100 100111	I	..XX			vmsumuhs	v2.03			345	Vector Multiply-Sum Unsigned Halfword Saturate VA-form
000100 .... ///// 00000 000001	I	..XX			vmul10cuq	v3.0			474	Vector Multiply-by-10 & write Carry-out Unsigned Quadword VX-form

Table G.1: Power ISA Instruction Set Sorted by Compliancy Subset (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
000100 .... 00001 000001	I	..XX			vmul10ecuq	v3.0			475	Vector Multiply-by-10 Extended & write Carry-out Unsigned Quadword VX-form
000100 .... 01001 000001	I	..XX			vmul10euq	v3.0			475	Vector Multiply-by-10 Extended Unsigned Quadword VX-form
000100 .... ///// 01000 000001	I	..XX			vmul10uq	v3.0			474	Vector Multiply-by-10 Unsigned Quadword VX-form
000100 .... 01100 001000	I	..XX			vmulesb	v2.03			329	Vector Multiply Even Signed Byte VX-form
000100 .... 01111 001000	I	..XX			vmulesd	v3.1			336	Vector Multiply Even Signed Doubleword VX-form
000100 .... 01101 001000	I	..XX			vmulesh	v2.03			331	Vector Multiply Even Signed Halfword VX-form
000100 .... 01110 001000	I	..XX			vmulesw	v2.07			333	Vector Multiply Even Signed Word VX-form
000100 .... 01000 001000	I	..XX			vmuleub	v2.03			330	Vector Multiply Even Unsigned Byte VX-form
000100 .... 01011 001000	I	..XX			vmuleud	v3.1			335	Vector Multiply Even Unsigned Doubleword VX-form
000100 .... 01001 001000	I	..XX			vmuleuh	v2.03			332	Vector Multiply Even Unsigned Halfword VX-form
000100 .... 01010 001000	I	..XX			vmuleuw	v2.07			334	Vector Multiply Even Unsigned Word VX-form
000100 .... 01111 001001	I	..XX			vmulhsd	v3.1			339	Vector Multiply High Signed Doubleword VX-form
000100 .... 01110 001001	I	..XX			vmulhsw	v3.1			338	Vector Multiply High Signed Word VX-form
000100 .... 01011 001001	I	..XX			vmulhud	v3.1			339	Vector Multiply High Unsigned Doubleword VX-form
000100 .... 01010 001001	I	..XX			vmulhuw	v3.1			338	Vector Multiply High Unsigned Word VX-form
000100 .... 00111 001001	I	..XX			vmulld	v3.1			340	Vector Multiply Low Doubleword VX-form
000100 .... 00100 001000	I	..XX			vmulosb	v2.03			329	Vector Multiply Odd Signed Byte VX-form
000100 .... 00111 001000	I	..XX			vmulosd	v3.1			336	Vector Multiply Odd Signed Doubleword VX-form
000100 .... 00101 001000	I	..XX			vmulosh	v2.03			331	Vector Multiply Odd Signed Halfword VX-form
000100 .... 00110 001000	I	..XX			vmulosw	v2.07			333	Vector Multiply Odd Signed Word VX-form
000100 .... 00000 001000	I	..XX			vmuloub	v2.03			330	Vector Multiply Odd Unsigned Byte VX-form
000100 .... 00011 001000	I	..XX			vmuloud	v3.1			335	Vector Multiply Odd Unsigned Doubleword VX-form
000100 .... 00001 001000	I	..XX			vmulouh	v2.03			332	Vector Multiply Odd Unsigned Halfword VX-form
000100 .... 00010 001000	I	..XX			vmulouw	v2.07			334	Vector Multiply Odd Unsigned Word VX-form
000100 .... 00010 001001	I	..XX			vmuluwm	v2.07			337	Vector Multiply Unsigned Word Modulo VX-form
000100 .... 10110 000100	I	..XX			vnand	v2.07			393	Vector Logical NAND VX-form
000100 .... 10101 001000	I	..XX			vncipher	v2.07			425	Vector AES Inverse Cipher VX-form
000100 .... 10101 001001	I	..XX			vncipherlast	v2.07			425	Vector AES Inverse Cipher Last VX-form
000100 ..... 00111 ..... 11000 000010	I	..XX			vnegd	v3.0			361	Vector Negate Doubleword VX-form
000100 ..... 00110 ..... 11000 000010	I	..XX			vnegw	v3.0			361	Vector Negate Word VX-form
000100 101111	I	..XX			vnmsubfp	v2.03			412	Vector Negative Multiply-Subtract Floating-Point VA-form
000100 .... 10100 000100	I	..XX			vnor	v2.03			394	Vector Logical NOR VX-form
000100 .... 10010 000100	I	..XX			vor	v2.03			393	Vector Logical OR VX-form
000100 .... 10101 000100	I	..XX			vorc	v2.07			393	Vector Logical OR with Complement VX-form
000100 .... 10111 001101	I	..XX			vpdepd	v3.1			442	Vector Parallel Bits Deposit Doubleword VX-form
000100 101011	I	..XX			vperm	v2.03			286	Vector Permute VA-form
000100 111011	I	..XX			vpermr	v3.0			286	Vector Permute Right-indexed VA-form
000100 101101	I	..XX			vpermxor	v2.07			432	Vector Permute & Exclusive-OR VA-form
000100 .... 10110 001101	I	..XX			vpextd	v3.1			443	Vector Parallel Bits Extract Doubleword VX-form
000100 .... 01100 001110	I	..XX			vpkpx	v2.03			268	Vector Pack Pixel VX-form
000100 .... 10111 001110	I	..XX			vpksdss	v2.07			271	Vector Pack Signed Doubleword Signed Saturate VX-form
000100 .... 10101 001110	I	..XX			vpksdus	v2.07			271	Vector Pack Signed Doubleword Unsigned Saturate VX-form
000100 .... 00110 001110	I	..XX			vpkshss	v2.03			269	Vector Pack Signed Halfword Signed Saturate VX-form
000100 .... 00100 001110	I	..XX			vpkshus	v2.03			269	Vector Pack Signed Halfword Unsigned Saturate VX-form
000100 .... 00111 001110	I	..XX			vpkswss	v2.03			270	Vector Pack Signed Word Signed Saturate VX-form

Table G.1: Power ISA Instruction Set Sorted by Compliancy Subset (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
000100 .... 00101 001110	I	..XX			vpkswus	v2.03			270	Vector Pack Signed Word Unsigned Saturate VX-form
000100 .... 10001 001110	I	..XX			vpkudum	v2.07			274	Vector Pack Unsigned Doubleword Unsigned Modulo VX-form
000100 .... 10011 001110	I	..XX			vpkudus	v2.07			274	Vector Pack Unsigned Doubleword Unsigned Saturate VX-form
000100 .... 00000 001110	I	..XX			vpkuhum	v2.03			272	Vector Pack Unsigned Halfword Unsigned Modulo VX-form
000100 .... 00010 001110	I	..XX			vpkuhus	v2.03			272	Vector Pack Unsigned Halfword Unsigned Saturate VX-form
000100 .... 00001 001110	I	..XX			vpkuwum	v2.03			273	Vector Pack Unsigned Word Unsigned Modulo VX-form
000100 .... 00011 001110	I	..XX			vpkuwus	v2.03			273	Vector Pack Unsigned Word Unsigned Saturate VX-form
000100 .... 10000 001000	I	..XX			vpmsumb	v2.07			430	Vector Polynomial Multiply-Sum Byte VX-form
000100 .... 10011 001000	I	..XX			vpmsumd	v2.07			431	Vector Polynomial Multiply-Sum Doubleword VX-form
000100 .... 10001 001000	I	..XX			vpmsumh	v2.07			430	Vector Polynomial Multiply-Sum Halfword VX-form
000100 .... 10010 001000	I	..XX			vpmsumw	v2.07			431	Vector Polynomial Multiply-Sum Word VX-form
000100 ..... ///// 11100 000011	I	..XX			vpopcntb	v2.07			445	Vector Population Count Byte VX-form
000100 ..... ///// 11111 000011	I	..XX			vpopcntd	v2.07			446	Vector Population Count Doubleword VX-form
000100 ..... ///// 11101 000011	I	..XX			vpopcnth	v2.07			445	Vector Population Count Halfword VX-form
000100 ..... ///// 11110 000011	I	..XX			vpopcntw	v2.07			446	Vector Population Count Word VX-form
000100 ..... 01001 ..... 11000 000010	I	..XX			vprtybd	v3.0			447	Vector Parity Byte Doubleword VX-form
000100 ..... 01010 ..... 11000 000010	I	..XX			vprtybq	v3.0			448	Vector Parity Byte Quadword VX-form
000100 ..... 01000 ..... 11000 000010	I	..XX			vprtybw	v3.0			447	Vector Parity Byte Word VX-form
000100 ..... ///// 00100 001010	I	..XX			vrefp	v2.03			423	Vector Reciprocal Estimate Floating-Point VX-form
000100 ..... ///// 01011 001010	I	..XX			vrfim	v2.03			416	Vector Round to Floating-Point Integer toward -Infinity VX-form
000100 ..... ///// 01000 001010	I	..XX			vrfin	v2.03			416	Vector Round to Floating-Point Integer Nearest VX-form
000100 ..... ///// 01010 001010	I	..XX			vrfip	v2.03			417	Vector Round to Floating-Point Integer toward +Infinity VX-form
000100 ..... ///// 01001 001010	I	..XX			vrfiz	v2.03			417	Vector Round to Floating-Point Integer toward Zero VX-form
000100 .... 00000 000100	I	..XX			vrlb	v2.03			395	Vector Rotate Left Byte VX-form
000100 .... 00011 000100	I	..XX			vrld	v2.07			396	Vector Rotate Left Doubleword VX-form
000100 .... 00011 000101	I	..XX			vrldmi	v3.0			401	Vector Rotate Left Doubleword then Mask Insert VX-form
000100 .... 00111 000101	I	..XX			vrldnm	v3.0			398	Vector Rotate Left Doubleword then AND with Mask VX-form
000100 .... 00001 000100	I	..XX			vrlh	v2.03			395	Vector Rotate Left Halfword VX-form
000100 .... 00000 000101	I	..XX			vrlq	v3.1			397	Vector Rotate Left Quadword VX-form
000100 .... 00001 000101	I	..XX			vrlqmi	v3.1			401	Vector Rotate Left Quadword then Mask Insert VX-form
000100 .... 00101 000101	I	..XX			vrlqnm	v3.1			399	Vector Rotate Left Quadword then AND with Mask VX-form
000100 .... 00010 000100	I	..XX			vrlw	v2.03			396	Vector Rotate Left Word VX-form
000100 .... 00010 000101	I	..XX			vrlwmi	v3.0			400	Vector Rotate Left Word then Mask Insert VX-form
000100 .... 00110 000101	I	..XX			vrlwnm	v3.0			398	Vector Rotate Left Word then AND with Mask VX-form
000100 ..... ///// 00101 001010	I	..XX			vrsqrtefp	v2.03			423	Vector Reciprocal Square Root Estimate Floating-Point VX-form
000100 .... ///// 10111 001000	I	..XX			vsbox	v2.07			426	Vector AES SubBytes VX-form
000100 101010	I	..XX			vsel	v2.03			287	Vector Select VA-form
000100 .... 11011 000010	I	..XX			vshasigmad	v2.07			427	Vector SHA-512 Sigma Doubleword VX-form
000100 .... 11010 000010	I	..XX			vshasigmaw	v2.07			428	Vector SHA-256 Sigma Word VX-form

Table G.1: Power ISA Instruction Set Sorted by Compliancy Subset (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
000100 .... 00111 000100	I	..XX			vsl	v2.03			290	Vector Shift Left VX-form
000100 .... 00100 000100	I	..XX			vslb	v2.03			402	Vector Shift Left Byte VX-form
000100 .... 10111 000100	I	..XX			vsld	v2.07			403	Vector Shift Left Doubleword VX-form
000100 .... 00... 010110	I	..XX			vsldbi	v3.1			289	Vector Shift Left Double by Bit Immediate VN-form
000100 .... /.... 101100	I	..XX			vsldoi	v2.03			289	Vector Shift Left Double by Octet Immediate VA-form
000100 .... 00101 000100	I	..XX			vslh	v2.03			402	Vector Shift Left Halfword VX-form
000100 .... 10000 001100	I	..XX			vslo	v2.03			291	Vector Shift Left by Octet VX-form
000100 .... 00100 000101	I	..XX			vslq	v3.1			404	Vector Shift Left Quadword VX-form
000100 .... 11101 000100	I	..XX			vslv	v3.0			292	Vector Shift Left Variable VX-form
000100 .... 00110 000100	I	..XX			vslw	v2.03			403	Vector Shift Left Word VX-form
000100 ..... /.... ..... 01000 001100	I	..XX			vspltb	v2.03			283	Vector Splat Byte VX-form
000100 ..... //... ..... 01001 001100	I	..XX			vsplth	v2.03			283	Vector Splat Halfword VX-form
000100 .... ///// 01100 001100	I	..XX			vspltisb	v2.03			285	Vector Splat Immediate Signed Byte VX-form
000100 .... ///// 01101 001100	I	..XX			vspltish	v2.03			285	Vector Splat Immediate Signed Halfword VX-form
000100 .... ///// 01110 001100	I	..XX			vspltisw	v2.03			285	Vector Splat Immediate Signed Word VX-form
000100 ..... ///.. ..... 01010 001100	I	..XX			vspltw	v2.03			284	Vector Splat Word VX-form
000100 .... 01011 000100	I	..XX			vsr	v2.03			290	Vector Shift Right VX-form
000100 .... 01100 000100	I	..XX			vsrab	v2.03			408	Vector Shift Right Algebraic Byte VX-form
000100 .... 01111 000100	I	..XX			vsrad	v2.07			409	Vector Shift Right Algebraic Doubleword VX-form
000100 .... 01101 000100	I	..XX			vsrah	v2.03			408	Vector Shift Right Algebraic Halfword VX-form
000100 .... 01100 000101	I	..XX			vsraq	v3.1			410	Vector Shift Right Algebraic Quadword VX-form
000100 .... 01110 000100	I	..XX			vsraw	v2.03			409	Vector Shift Right Algebraic Word VX-form
000100 .... 01000 000100	I	..XX			vsrb	v2.03			405	Vector Shift Right Byte VX-form
000100 .... 11011 000100	I	..XX			vsrd	v2.07			406	Vector Shift Right Doubleword VX-form
000100 .... 01... 010110	I	..XX			vsrdbi	v3.1			289	Vector Shift Right Double by Bit Immediate VN-form
000100 .... 01001 000100	I	..XX			vsrh	v2.03			405	Vector Shift Right Halfword VX-form
000100 .... 10001 001100	I	..XX			vsro	v2.03			291	Vector Shift Right by Octet VX-form
000100 .... 01000 000101	I	..XX			vsrq	v3.1			407	Vector Shift Right Quadword VX-form
000100 .... 11100 000100	I	..XX			vsrv	v3.0			292	Vector Shift Right Variable VX-form
000100 .... 01010 000100	I	..XX			vsrw	v2.03			406	Vector Shift Right Word VX-form
000100 ..... 00000 ..... .0000 001101	I	..XX			vstribl[.]	v3.1			462	Vector String Isolate Byte Left-justified VC-form
000100 ..... 00001 ..... .0000 001101	I	..XX			vstribr[.]	v3.1			462	Vector String Isolate Byte Right-justified VC-form
000100 ..... 00010 ..... .0000 001101	I	..XX			vstrihl[.]	v3.1			463	Vector String Isolate Halfword Left-justified VC-form
000100 ..... 00011 ..... .0000 001101	I	..XX			vstrihr[.]	v3.1			463	Vector String Isolate Halfword Right-justified VC-form
000100 .... 10101 000000	I	..XX			vsubcuq	v2.07			328	Vector Subtract & write Carry-out Unsigned Quadword VX-form
000100 .... 10110 000000	I	..XX			vsubcuw	v2.03			321	Vector Subtract & Write Carry-out Unsigned Word VX-form
000100 111111	I	..XX			vsubecuq	v2.07			328	Vector Subtract Extended & write Carry-out Unsigned Quadword VA-form
000100 111110	I	..XX			vsubeuqm	v2.07			327	Vector Subtract Extended Unsigned Quadword Modulo VA-form
000100 .... 00001 001010	I	..XX			vsubfp	v2.03			411	Vector Subtract Floating-Point VX-form
000100 .... 11100 000000	I	..XX			vsubsbs	v2.03			321	Vector Subtract Signed Byte Saturate VX-form
000100 .... 11101 000000	I	..XX			vsubshs	v2.03			322	Vector Subtract Signed Halfword Saturate VX-form
000100 .... 11110 000000	I	..XX			vsubsws	v2.03			322	Vector Subtract Signed Word Saturate VX-form
000100 .... 10000 000000	I	..XX			vsububm	v2.03			323	Vector Subtract Unsigned Byte Modulo VX-form
000100 .... 11000 000000	I	..XX			vsububs	v2.03			325	Vector Subtract Unsigned Byte Saturate VX-form

Table G.1: Power ISA Instruction Set Sorted by Compliancy Subset (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
000100 .... 10011 000000	I	..XX			vsubudm	v2.07			324	Vector Subtract Unsigned Doubleword Modulo VX-form
000100 .... 10001 000000	I	..XX			vsubuhm	v2.03			323	Vector Subtract Unsigned Halfword Modulo VX-form
000100 .... 11001 000000	I	..XX			vsubuhs	v2.03			325	Vector Subtract Unsigned Halfword Saturate VX-form
000100 .... 10100 000000	I	..XX			vsubuqm	v2.07			327	Vector Subtract Unsigned Quadword Modulo VX-form
000100 .... 10010 000000	I	..XX			vsubuwm	v2.03			324	Vector Subtract Unsigned Word Modulo VX-form
000100 .... 11010 000000	I	..XX			vsubuws	v2.03			326	Vector Subtract Unsigned Word Saturate VX-form
000100 .... 11010 001000	I	..XX			vsum2sws	v2.03			358	Vector Sum across Half Signed Word Saturate VX-form
000100 .... 11100 001000	I	..XX			vsum4sbs	v2.03			359	Vector Sum across Quarter Signed Byte Saturate VX-form
000100 .... 11001 001000	I	..XX			vsum4shs	v2.03			359	Vector Sum across Quarter Signed Halfword Saturate VX-form
000100 .... 11000 001000	I	..XX			vsum4ubs	v2.03			360	Vector Sum across Quarter Unsigned Byte Saturate VX-form
000100 .... 11110 001000	I	..XX			vsumsws	v2.03			357	Vector Sum across Signed Word Saturate VX-form
000100 ..... ///// 01101 001110	I	..XX			vupkhpx	v2.03			278	Vector Unpack High Pixel VX-form
000100 ..... ///// 01000 001110	I	..XX			vupkhsb	v2.03			275	Vector Unpack High Signed Byte VX-form
000100 ..... ///// 01001 001110	I	..XX			vupkhsh	v2.03			276	Vector Unpack High Signed Halfword VX-form
000100 ..... ///// 11001 001110	I	..XX			vupkhsw	v2.07			277	Vector Unpack High Signed Word VX-form
000100 ..... ///// 01111 001110	I	..XX			vupklpx	v2.03			278	Vector Unpack Low Pixel VX-form
000100 ..... ///// 01010 001110	I	..XX			vupklsb	v2.03			275	Vector Unpack Low Signed Byte VX-form
000100 ..... ///// 01011 001110	I	..XX			vupklsh	v2.03			276	Vector Unpack Low Signed Halfword VX-form
000100 ..... ///// 11011 001110	I	..XX			vupklsw	v2.07			277	Vector Unpack Low Signed Word VX-form
000100 .... 10011 000100	I	..XX			vxor	v2.03			394	Vector Logical XOR VX-form
011111 ///.. ///.. ///// 00000 11110/	II	..XX			wait	v2.03			1029	Wait X-form
111100 ..... ///// ..... 10101 1001..	I	..XX			xsabsdp	v2.06			607	VSX Scalar Absolute Double-Precision XX2-form
111100 .... 00100 000...	I	..XX			xsadddp	v2.06			615	VSX Scalar Add Double-Precision XX3-form
111100 .... 00000 000...	I	..XX			xsaddsp	v2.07			622	VSX Scalar Add Single-Precision XX3-form
111100 .... 00000 011...	I	..XX			xscmpeqdp	v3.0			749	VSX Scalar Compare Equal Double-Precision XX3-form
111111 .... 00010 00100/	I	..XX			xscmpeqqp	v3.1			750	VSX Scalar Compare Equal Quad-Precision X-form
111100 ...// .... 00111 011../	I	..XX			xscmpexpdp	v3.0			862	VSX Scalar Compare Exponents Double-Precision XX3-form
111100 .... 00010 011...	I	..XX			xscmpgedp	v3.0			751	VSX Scalar Compare Greater Than or Equal Double-Precision XX3-form
111111 .... 00110 00100/	I	..XX			xscmpgeqp	v3.1			752	VSX Scalar Compare Greater Than or Equal Quad-Precision X-form
111100 .... 00001 011...	I	..XX			xscmpgtdp	v3.0			753	VSX Scalar Compare Greater Than Double-Precision XX3-form
111111 .... 00111 00100/	I	..XX			xscmpgtqp	v3.1			754	VSX Scalar Compare Greater Than Quad-Precision X-form
111100 ...// .... 00101 011../	I	..XX			xscmpodp	v2.06			743	VSX Scalar Compare Ordered Double-Precision XX3-form
111100 ...// .... 00100 011../	I	..XX			xscmpudp	v2.06			746	VSX Scalar Compare Unordered Double-Precision XX3-form
111100 .... 10110 000...	I	..XX			xscpsgndp	v2.06			608	VSX Scalar Copy Sign Double-Precision XX3-form
111100 ..... 10001 ..... 10101 1011..	I	..XX			xscvdphp	v3.0			788	VSX Scalar Convert with round Double-Precision to Half-Precision format XX2-form
111100 ..... ///// ..... 10000 1001..	I	..XX			xscvdpsp	v2.06			789	VSX Scalar Convert with round Double-Precision to Single-Precision format XX2-form
111100 ..... ///// ..... 10000 1011..	I	..XX			xscvdpspn	v2.07			790	VSX Scalar Convert Scalar Single-Precision to Vector Single-Precision format Non-signalling XX2-form
111100 ..... ///// ..... 10101 1000..	I	..XX			xscvdpsxds	v2.06			816	VSX Scalar Convert with round to zero Double-Precision to Signed Doubleword format XX2-form

Table G.1: Power ISA Instruction Set Sorted by Compliancy Subset (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
111100 ..... ///// ..... 00101 1000..	I	..XX			xscvdpsxws	v2.06			818	VSX Scalar Convert with round to zero Double-Precision to Signed Word format XX2-form
111100 ..... ///// ..... 10100 1000..	I	..XX			xscvdpuxds	v2.06			820	VSX Scalar Convert with round to zero Double-Precision to Unsigned Doubleword format XX2-form
111100 ..... ///// ..... 00100 1000..	I	..XX			xscvdpuxws	v2.06			822	VSX Scalar Convert with round to zero Double-Precision to Unsigned Word format XX2-form
111100 ..... 10000 ..... 10101 1011..	I	..XX			xscvhpdp	v3.0			796	VSX Scalar Convert Half-Precision to Double-Precision format XX2-form
111111 ..... 01000 ..... 11010 00100/	I	..XX			xscvqpsqz	v3.1			826	VSX Scalar Convert with round to zero Quad-Precision to Signed Quadword X-form
111111 ..... 00000 ..... 11010 00100/	I	..XX			xscvqpuqz	v3.1			832	VSX Scalar Convert with round to zero Quad-Precision to Unsigned Quadword X-form
111100 ..... ///// ..... 10100 1001..	I	..XX			xscvspdp	v2.06			797	VSX Scalar Convert Single-Precision to Double-Precision format XX2-form
111100 ..... ///// ..... 10100 1011..	I	..XX			xscvspdpn	v2.07			798	VSX Scalar Convert Single-Precision to Double-Precision format Non-signalling XX2-form
111111 ..... 01011 ..... 11010 00100/	I	..XX			xscvsqqp	v3.1			853	VSX Scalar Convert with round Signed Quadword to Quad-Precision X-form
111100 ..... ///// ..... 10111 1000..	I	..XX			xscvsxddp	v2.06			854	VSX Scalar Convert with round Signed Doubleword to Double-Precision format XX2-form
111100 ..... ///// ..... 10011 1000..	I	..XX			xscvsxdsp	v2.07			855	VSX Scalar Convert with round Signed Doubleword to Single-Precision format XX2-form
111111 ..... 00011 ..... 11010 00100/	I	..XX			xscvuqqp	v3.1			853	VSX Scalar Convert with round Unsigned Quadword to Quad-Precision format X-form
111100 ..... ///// ..... 10110 1000..	I	..XX			xscvuxddp	v2.06			854	VSX Scalar Convert with round Unsigned Doubleword to Double-Precision format XX2-form
111100 ..... ///// ..... 10010 1000..	I	..XX			xscvuxdsp	v2.07			855	VSX Scalar Convert with round Unsigned Doubleword to Single-Precision format XX2-form
111100 .... 00111 000...	I	..XX			xsdivdp	v2.06			624	VSX Scalar Divide Double-Precision XX3-form
111100 .... 00011 000...	I	..XX			xsdivsp	v2.07			628	VSX Scalar Divide Single-Precision XX3-form
111100 .... 11100 10110.	I	..XX			xsiexpdp	v3.0			864	VSX Scalar Insert Exponent Double-Precision X-form
111100 .... 00100 001...	I	..XX			xsmaddadp	v2.06			648	VSX Scalar Multiply-Add Type-A Double-Precision XX3-form
111100 .... 00000 001...	I	..XX			xsmaddasp	v2.07			651	VSX Scalar Multiply-Add Type-A Single-Precision XX3-form
111100 .... 00101 001...	I	..XX			xsmaddmdp	v2.06			648	VSX Scalar Multiply-Add Type-M Double-Precision XX3-form
111100 .... 00001 001...	I	..XX			xsmaddmsp	v2.07			651	VSX Scalar Multiply-Add Type-M Single-Precision XX3-form
111100 .... 10000 000...	I	..XX			xsmaxcdp	v3.0			755	VSX Scalar Maximum Type-C Double-Precision XX3-form
111111 .... 10101 00100/	I	..XX			xsmaxcqp	v3.1			757	VSX Scalar Maximum Type-C Quad-Precision X-form
111100 .... 10100 000...	I	..XX			xsmaxdp	v2.06			759	VSX Scalar Maximum Double-Precision XX3-form
111100 .... 10010 000...	I	..XX			xsmaxjdp	v3.0			761	VSX Scalar Maximum Type-J Double-Precision XX3-form
111100 .... 10001 000...	I	..XX			xsmincdp	v3.0			763	VSX Scalar Minimum Type-C Double-Precision XX3-form
111111 .... 10111 00100/	I	..XX			xsmincqp	v3.1			765	VSX Scalar Minimum Type-C Quad-Precision X-form
111100 .... 10101 000...	I	..XX			xsmindp	v2.06			767	VSX Scalar Minimum Double-Precision XX3-form
111100 .... 10011 000...	I	..XX			xsminjdp	v3.0			769	VSX Scalar Minimum Type-J Double-Precision XX3-form
111100 .... 00110 001...	I	..XX			xsmsubadp	v2.06			657	VSX Scalar Multiply-Subtract Type-A Double-Precision XX3-form

Table G.1: Power ISA Instruction Set Sorted by Compliancy Subset (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
111100 .... 00010 001...	I	..XX			xsmsubasp	v2.07			660	VSX Scalar Multiply-Subtract Type-A Single-Precision XX3-form
111100 .... 00111 001...	I	..XX			xsmsubmdp	v2.06			657	VSX Scalar Multiply-Subtract Type-M Double-Precision XX3-form
111100 .... 00011 001...	I	..XX			xsmsubmsp	v2.07			660	VSX Scalar Multiply-Subtract Type-M Single-Precision XX3-form
111100 .... 00110 000...	I	..XX			xsmuldp	v2.06			630	VSX Scalar Multiply Double-Precision XX3-form
111100 .... 00010 000...	I	..XX			xsmulsp	v2.07			634	VSX Scalar Multiply Single-Precision XX3-form
111100 ..... ///// ..... 10110 1001..	I	..XX			xsnabsdp	v2.06			609	VSX Scalar Negative Absolute Double-Precision XX2-form
111100 ..... ///// ..... 10111 1001..	I	..XX			xsnegdp	v2.06			610	VSX Scalar Negate Double-Precision XX2-form
111100 .... 10100 001...	I	..XX			xsnmaddadp	v2.06			666	VSX Scalar Negative Multiply-Add Type-A Double-Precision XX3-form
111100 .... 10000 001...	I	..XX			xsnmaddasp	v2.07			670	VSX Scalar Negative Multiply-Add Type-A Single-Precision XX3-form
111100 .... 10101 001...	I	..XX			xsnmaddmdp	v2.06			666	VSX Scalar Negative Multiply-Add Type-M Double-Precision XX3-form
111100 .... 10001 001...	I	..XX			xsnmaddmsp	v2.07			670	VSX Scalar Negative Multiply-Add Type-M Single-Precision XX3-form
111100 .... 10110 001...	I	..XX			xsnmsubadp	v2.06			676	VSX Scalar Negative Multiply-Subtract Type-A Double-Precision XX3-form
111100 .... 10010 001...	I	..XX			xsnmsubasp	v2.07			679	VSX Scalar Negative Multiply-Subtract Type-A Single-Precision XX3-form
111100 .... 10111 001...	I	..XX			xsnmsubmdp	v2.06			676	VSX Scalar Negative Multiply-Subtract Type-M Double-Precision XX3-form
111100 .... 10011 001...	I	..XX			xsnmsubmsp	v2.07			679	VSX Scalar Negative Multiply-Subtract Type-M Single-Precision XX3-form
111100 ..... ///// ..... 00100 1001..	I	..XX			xsrdpi	v2.06			801	VSX Scalar Round to Double-Precision Integer using round to Nearest Away XX2-form
111100 ..... ///// ..... 00110 1011..	I	..XX			xsrdpic	v2.06			802	VSX Scalar Round to Double-Precision Integer exact using Current rounding mode XX2-form
111100 ..... ///// ..... 00111 1001..	I	..XX			xsrdpim	v2.06			803	VSX Scalar Round to Double-Precision Integer using round toward -Infinity XX2-form
111100 ..... ///// ..... 00110 1001..	I	..XX			xsrdpip	v2.06			804	VSX Scalar Round to Double-Precision Integer using round toward +Infinity XX2-form
111100 ..... ///// ..... 00101 1001..	I	..XX			xsrdpiz	v2.06			805	VSX Scalar Round to Double-Precision Integer using round toward Zero XX2-form
111100 ..... ///// ..... 00101 1010..	I	..XX			xsredp	v2.06			685	VSX Scalar Reciprocal Estimate Double-Precision XX2-form
111100 ..... ///// ..... 00001 1010..	I	..XX			xsresp	v2.07			686	VSX Scalar Reciprocal Estimate Single-Precision XX2-form
111100 ..... ///// ..... 10001 1001..	I	..XX			xsrsp	v2.07			787	VSX Scalar Round to Single-Precision XX2-form
111100 ..... ///// ..... 00100 1010..	I	..XX			xsrsqrtedp	v2.06			687	VSX Scalar Reciprocal Square Root Estimate Double-Precision XX2-form
111100 ..... ///// ..... 00000 1010..	I	..XX			xsrsqrtesp	v2.07			688	VSX Scalar Reciprocal Square Root Estimate Single-Precision XX2-form
111100 ..... ///// ..... 00100 1011..	I	..XX			xssqrtdp	v2.06			636	VSX Scalar Square Root Double-Precision XX2-form
111100 ..... ///// ..... 00000 1011..	I	..XX			xssqrtsp	v2.07			640	VSX Scalar Square Root Single-Precision XX2-form
111100 .... 00101 000...	I	..XX			xssubdp	v2.06			642	VSX Scalar Subtract Double-Precision XX3-form
111100 .... 00001 000...	I	..XX			xssubsp	v2.07			646	VSX Scalar Subtract Single-Precision XX3-form

Table G.1: Power ISA Instruction Set Sorted by Compliancy Subset (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
111100 ...// .... 00111 101../	I	..XX			xstdivdp	v2.06			689	VSX Scalar Test for software Divide Double-Precision XX3-form
111100 ...// ///// 00110 1010./	I	..XX			xstsqrtdp	v2.06			690	VSX Scalar Test for software Square Root Double-Precision XX2-form
111100 .... 10110 1010./	I	..XX			xststdcdp	v3.0			866	VSX Scalar Test Data Class Double-Precision XX2-form
111100 .... 10010 1010./	I	..XX			xststdcsp	v3.0			868	VSX Scalar Test Data Class Single-Precision XX2-form
111100 ..... 00000 ..... 10101 1011./	I	..XX			xsxexpdp	v3.0			869	VSX Scalar Extract Exponent Double-Precision XX2-form
111100 ..... 00001 ..... 10101 1011./	I	..XX			xsxsigdp	v3.0			870	VSX Scalar Extract Significand Double-Precision XX2-form
111100 ..... ///// ..... 11101 1001..	I	..XX			xvabsdp	v2.06			611	VSX Vector Absolute Double-Precision XX2-form
111100 ..... ///// ..... 11001 1001..	I	..XX			xvabssp	v2.06			611	VSX Vector Absolute Single-Precision XX2-form
111100 .... 01100 000...	I	..XX			xvadddp	v2.06			691	VSX Vector Add Double-Precision XX3-form
111100 .... 01000 000...	I	..XX			xvaddsp	v2.06			694	VSX Vector Add Single-Precision XX3-form
111100 .... .1100 011...	I	..XX			xvcmpeqdp[.]	v2.06			771	VSX Vector Compare Equal To Double-Precision XX3-form
111100 .... .1000 011...	I	..XX			xvcmpeqsp[.]	v2.06			772	VSX Vector Compare Equal To Single-Precision XX3-form
111100 .... .1110 011...	I	..XX			xvcmpgedp[.]	v2.06			773	VSX Vector Compare Greater Than or Equal To Double-Precision XX3-form
111100 .... .1010 011...	I	..XX			xvcmpgesp[.]	v2.06			774	VSX Vector Compare Greater Than or Equal To Single-Precision XX3-form
111100 .... .1101 011...	I	..XX			xvcmpgtdp[.]	v2.06			775	VSX Vector Compare Greater Than Double-Precision XX3-form
111100 .... .1001 011...	I	..XX			xvcmpgtsp[.]	v2.06			776	VSX Vector Compare Greater Than Single-Precision XX3-form
111100 .... 11110 000...	I	..XX			xvcpsgndp	v2.06			612	VSX Vector Copy Sign Double-Precision XX3-form
111100 .... 11010 000...	I	..XX			xvcpsgnsp	v2.06			612	VSX Vector Copy Sign Single-Precision XX3-form
111100 ..... 10000 ..... 11101 1011..	I	..XX			xvcvbf16spn	v3.1			800	VSX Vector Convert bfloat16 to Single-Precision format Non-signaling XX2-form
111100 ..... ///// ..... 11000 1001..	I	..XX			xvcvdpsp	v2.06			793	VSX Vector Convert with round Double-Precision to Single-Precision format XX2-form
111100 ..... ///// ..... 11101 1000..	I	..XX			xvcvdpsxds	v2.06			836	VSX Vector Convert with round to zero Double-Precision to Signed Doubleword format XX2-form
111100 ..... ///// ..... 01101 1000..	I	..XX			xvcvdpsxws	v2.06			838	VSX Vector Convert with round to zero Double-Precision to Signed Word format XX2-form
111100 ..... ///// ..... 11100 1000..	I	..XX			xvcvdpuxds	v2.06			840	VSX Vector Convert with round to zero Double-Precision to Unsigned Doubleword format XX2-form
111100 ..... ///// ..... 01100 1000..	I	..XX			xvcvdpuxws	v2.06			842	VSX Vector Convert with round to zero Double-Precision to Unsigned Word format XX2-form
111100 ..... 11000 ..... 11101 1011..	I	..XX			xvcvhpsp	v3.0			799	VSX Vector Convert Half-Precision to Single-Precision format XX2-form
111100 ..... 10001 ..... 11101 1011..	I	..XX			xvcvspbf16	v3.1			792	VSX Vector Convert with round Single-Precision to bfloat16 format XX2-form
111100 ..... ///// ..... 11100 1001..	I	..XX			xvcvspdp	v2.06			800	VSX Vector Convert Single-Precision to Double-Precision format XX2-form
111100 ..... 11001 ..... 11101 1011..	I	..XX			xvcvsphp	v3.0			794	VSX Vector Convert with round Single-Precision to Half-Precision format XX2-form
111100 ..... ///// ..... 11001 1000..	I	..XX			xvcvspsxds	v2.06			844	VSX Vector Convert with round to zero Single-Precision to Signed Doubleword format XX2-form
111100 ..... ///// ..... 01001 1000..	I	..XX			xvcvspsxws	v2.06			846	VSX Vector Convert with round to zero Single-Precision to Signed Word format XX2-form

Table G.1: Power ISA Instruction Set Sorted by Compliancy Subset (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
111100 ..... ///// ..... 11000 1000..	I	..XX			xvcvspuxds	v2.06			848	VSX Vector Convert with round to zero Single-Precision to Unsigned Doubleword format XX2-form
111100 ..... ///// ..... 01000 1000..	I	..XX			xvcvspuxws	v2.06			850	VSX Vector Convert with round to zero Single-Precision to Unsigned Word format XX2-form
111100 ..... ///// ..... 11111 1000..	I	..XX			xvcvsxddp	v2.06			857	VSX Vector Convert with round Signed Doubleword to Double-Precision format XX2-form
111100 ..... ///// ..... 11011 1000..	I	..XX			xvcvsxdsp	v2.06			859	VSX Vector Convert with round Signed Doubleword to Single-Precision format XX2-form
111100 ..... ///// ..... 01111 1000..	I	..XX			xvcvsxwdp	v2.06			858	VSX Vector Convert Signed Word to Double-Precision format XX2-form
111100 ..... ///// ..... 01011 1000..	I	..XX			xvcvsxwsp	v2.06			861	VSX Vector Convert with round Signed Word to Single-Precision format XX2-form
111100 ..... ///// ..... 11110 1000..	I	..XX			xvcvuxddp	v2.06			857	VSX Vector Convert with round Unsigned Doubleword to Double-Precision format XX2-form
111100 ..... ///// ..... 11010 1000..	I	..XX			xvcvuxdsp	v2.06			859	VSX Vector Convert with round Unsigned Doubleword to Single-Precision format XX2-form
111100 ..... ///// ..... 01110 1000..	I	..XX			xvcvuxwdp	v2.06			858	VSX Vector Convert Unsigned Word to Double-Precision format XX2-form
111100 ..... ///// ..... 01010 1000..	I	..XX			xvcvuxwsp	v2.06			861	VSX Vector Convert with round Unsigned Word to Single-Precision format XX2-form
111100 .... 01111 000...	I	..XX			xvdivdp	v2.06			696	VSX Vector Divide Double-Precision XX3-form
111100 .... 01011 000...	I	..XX			xvdivsp	v2.06			698	VSX Vector Divide Single-Precision XX3-form
111100 .... 11111 000...	I	..XX			xviexpdp	v3.0			871	VSX Vector Insert Exponent Double-Precision XX3-form
111100 .... 11011 000...	I	..XX			xviexpsp	v3.0			871	VSX Vector Insert Exponent Single-Precision XX3-form
111100 .... 01100 001...	I	..XX			xvmaddadp	v2.06			710	VSX Vector Multiply-Add Type-A Double-Precision XX3-form
111100 .... 01000 001...	I	..XX			xvmaddasp	v2.06			713	VSX Vector Multiply-Add Type-A Single-Precision XX3-form
111100 .... 01101 001...	I	..XX			xvmaddmdp	v2.06			710	VSX Vector Multiply-Add Type-M Double-Precision XX3-form
111100 .... 01001 001...	I	..XX			xvmaddmsp	v2.06			713	VSX Vector Multiply-Add Type-M Single-Precision XX3-form
111100 .... 11100 000...	I	..XX			xvmaxdp	v2.06			777	VSX Vector Maximum Double-Precision XX3-form
111100 .... 11000 000...	I	..XX			xvmaxsp	v2.06			779	VSX Vector Maximum Single-Precision XX3-form
111100 .... 11101 000...	I	..XX			xvmindp	v2.06			781	VSX Vector Minimum Double-Precision XX3-form
111100 .... 11001 000...	I	..XX			xvminsp	v2.06			783	VSX Vector Minimum Single-Precision XX3-form
111100 .... 01110 001...	I	..XX			xvmsubadp	v2.06			716	VSX Vector Multiply-Subtract Type-A Double-Precision XX3-form
111100 .... 01010 001...	I	..XX			xvmsubasp	v2.06			719	VSX Vector Multiply-Subtract Type-A Single-Precision XX3-form
111100 .... 01111 001...	I	..XX			xvmsubmdp	v2.06			716	VSX Vector Multiply-Subtract Type-M Double-Precision XX3-form
111100 .... 01011 001...	I	..XX			xvmsubmsp	v2.06			719	VSX Vector Multiply-Subtract Type-M Single-Precision XX3-form
111100 .... 01110 000...	I	..XX			xvmuldp	v2.06			700	VSX Vector Multiply Double-Precision XX3-form
111100 .... 01010 000...	I	..XX			xvmulsp	v2.06			702	VSX Vector Multiply Single-Precision XX3-form
111100 ..... ///// ..... 11110 1001..	I	..XX			xvnabsdp	v2.06			613	VSX Vector Negative Absolute Double-Precision XX2-form
111100 ..... ///// ..... 11010 1001..	I	..XX			xvnabssp	v2.06			613	VSX Vector Negative Absolute Single-Precision XX2-form
111100 ..... ///// ..... 11111 1001..	I	..XX			xvnegdp	v2.06			614	VSX Vector Negate Double-Precision XX2-form

Table G.1: Power ISA Instruction Set Sorted by Compliancy Subset (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
111100 ..... ///// ..... 11011 1001..	I	..XX			xvnegsp	v2.06			614	VSX Vector Negate Single-Precision XX2-form
111100 .... 11100 001...	I	..XX			xvnmaddadp	v2.06			722	VSX Vector Negative Multiply-Add Type-A Double-Precision XX3-form
111100 .... 11000 001...	I	..XX			xvnmaddasp	v2.06			726	VSX Vector Negative Multiply-Add Type-A Single-Precision XX3-form
111100 .... 11101 001...	I	..XX			xvnmaddmdp	v2.06			722	VSX Vector Negative Multiply-Add Type-M Double-Precision XX3-form
111100 .... 11001 001...	I	..XX			xvnmaddmsp	v2.06			726	VSX Vector Negative Multiply-Add Type-M Single-Precision XX3-form
111100 .... 11110 001...	I	..XX			xvnmsubadp	v2.06			729	VSX Vector Negative Multiply-Subtract Type-A Double-Precision XX3-form
111100 .... 11010 001...	I	..XX			xvnmsubasp	v2.06			732	VSX Vector Negative Multiply-Subtract Type-A Single-Precision XX3-form
111100 .... 11111 001...	I	..XX			xvnmsubmdp	v2.06			729	VSX Vector Negative Multiply-Subtract Type-M Double-Precision XX3-form
111100 .... 11011 001...	I	..XX			xvnmsubmsp	v2.06			732	VSX Vector Negative Multiply-Subtract Type-M Single-Precision XX3-form
111100 ..... ///// ..... 01100 1001..	I	..XX			xvrdpi	v2.06			808	VSX Vector Round to Double-Precision Integer using round to Nearest Away XX2-form
111100 ..... ///// ..... 01110 1011..	I	..XX			xvrdpic	v2.06			809	VSX Vector Round to Double-Precision Integer Exact using Current rounding mode XX2-form
111100 ..... ///// ..... 01111 1001..	I	..XX			xvrdpim	v2.06			810	VSX Vector Round to Double-Precision Integer using round toward -Infinity XX2-form
111100 ..... ///// ..... 01110 1001..	I	..XX			xvrdpip	v2.06			810	VSX Vector Round to Double-Precision Integer using round toward +Infinity XX2-form
111100 ..... ///// ..... 01101 1001..	I	..XX			xvrdpiz	v2.06			811	VSX Vector Round to Double-Precision Integer using round toward Zero XX2-form
111100 ..... ///// ..... 01101 1010..	I	..XX			xvredp	v2.06			735	VSX Vector Reciprocal Estimate Double-Precision XX2-form
111100 ..... ///// ..... 01001 1010..	I	..XX			xvresp	v2.06			736	VSX Vector Reciprocal Estimate Single-Precision XX2-form
111100 ..... ///// ..... 01000 1001..	I	..XX			xvrspi	v2.06			812	VSX Vector Round to Single-Precision Integer using round to Nearest Away XX2-form
111100 ..... ///// ..... 01010 1011..	I	..XX			xvrspic	v2.06			813	VSX Vector Round to Single-Precision Integer Exact using Current rounding mode XX2-form
111100 ..... ///// ..... 01011 1001..	I	..XX			xvrspim	v2.06			814	VSX Vector Round to Single-Precision Integer using round toward -Infinity XX2-form
111100 ..... ///// ..... 01010 1001..	I	..XX			xvrspip	v2.06			814	VSX Vector Round to Single-Precision Integer using round toward +Infinity XX2-form
111100 ..... ///// ..... 01001 1001..	I	..XX			xvrspiz	v2.06			815	VSX Vector Round to Single-Precision Integer using round toward Zero XX2-form
111100 ..... ///// ..... 01100 1010..	I	..XX			xvrsqrtedp	v2.06			737	VSX Vector Reciprocal Square Root Estimate Double-Precision XX2-form
111100 ..... ///// ..... 01000 1010..	I	..XX			xvrsqrtesp	v2.06			738	VSX Vector Reciprocal Square Root Estimate Single-Precision XX2-form
111100 ..... ///// ..... 01100 1011..	I	..XX			xvsqrtdp	v2.06			704	VSX Vector Square Root Double-Precision XX2-form
111100 ..... ///// ..... 01000 1011..	I	..XX			xvsqrtsp	v2.06			705	VSX Vector Square Root Single-Precision XX2-form
111100 .... 01101 000...	I	..XX			xvsubdp	v2.06			706	VSX Vector Subtract Double-Precision XX3-form
111100 .... 01001 000...	I	..XX			xvsubsp	v2.06			708	VSX Vector Subtract Single-Precision XX3-form

Table G.1: Power ISA Instruction Set Sorted by Compliancy Subset (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
111100 ...// .... 01111 101../	I	..XX			xvtdivdp	v2.06			739	VSX Vector Test for software Divide Double-Precision XX3-form
111100 ...// .... 01011 101../	I	..XX			xvtdivsp	v2.06			740	VSX Vector Test for software Divide Single-Precision XX3-form
111100 ...// 00010 ..... 11101 1011./	I	..XX			xvtlsbb	v3.1			936	VSX Vector Test Least-Significant Bit by Byte XX2-form
111100 ...// ///// 01110 1010./	I	..XX			xvtsqrtdp	v2.06			741	VSX Vector Test for software Square Root Double-Precision XX2-form
111100 ...// ///// 01010 1010./	I	..XX			xvtsqrtsp	v2.06			742	VSX Vector Test for software Square Root Single-Precision XX2-form
111100 .... 1111. 101...	I	..XX			xvtstdcdp	v3.0			872	VSX Vector Test Data Class Double-Precision XX2-form
111100 .... 1101. 101...	I	..XX			xvtstdcsp	v3.0			873	VSX Vector Test Data Class Single-Precision XX2-form
111100 ..... 00000 ..... 11101 1011..	I	..XX			xvxexpdp	v3.0			874	VSX Vector Extract Exponent Double-Precision XX2-form
111100 ..... 01000 ..... 11101 1011..	I	..XX			xvxexpsp	v3.0			874	VSX Vector Extract Exponent Single-Precision XX2-form
111100 ..... 00001 ..... 11101 1011..	I	..XX			xvxsigdp	v3.0			875	VSX Vector Extract Significand Double-Precision XX2-form
111100 ..... 01001 ..... 11101 1011..	I	..XX			xvxsigsp	v3.0			875	VSX Vector Extract Significand Single-Precision XX2-form
000001 01000 0//// ///// ///// ////// 100001 .... 00....	I	..XX			xxblendvb	v3.1			912	VSX Vector Blend Variable Byte 8RR:XX4-form
000001 01000 0//// ///// ///// ////// 100001 .... 11....	I	..XX			xxblendvd	v3.1			912	VSX Vector Blend Variable Doubleword 8RR:XX4-form
000001 01000 0//// ///// ///// ////// 100001 .... 01....	I	..XX			xxblendvh	v3.1			913	VSX Vector Blend Variable Halfword 8RR:XX4-form
000001 01000 0//// ///// ///// ////// 100001 .... 10....	I	..XX			xxblendvw	v3.1			913	VSX Vector Blend Variable Word 8RR:XX4-form
111100 ..... 10111 ..... 11101 1011..	I	..XX			xxbrd	v3.0			914	VSX Vector Byte-Reverse Doubleword XX2-form
111100 ..... 00111 ..... 11101 1011..	I	..XX			xxbrh	v3.0			915	VSX Vector Byte-Reverse Halfword XX2-form
111100 ..... 11111 ..... 11101 1011..	I	..XX			xxbrq	v3.0			915	VSX Vector Byte-Reverse Quadword XX2-form
111100 ..... 01111 ..... 11101 1011..	I	..XX			xxbrw	v3.0			916	VSX Vector Byte-Reverse Word XX2-form
000001 01000 0//// ///// ///.. ..... 100010 .... 01....	I	..XX			xxeval	v3.1			910	VSX Vector Evaluate 8RR:XX4-form
111100 ..... /.... ..... 01010 0101..	I	..XX			xxextractuw	v3.0			917	VSX Vector Extract Unsigned Word XX2-form
111100 .... 11100 10100.	I	..XX			xxgenpcvbm	v3.1			926	VSX Vector Generate PCV from Byte Mask X-form
111100 .... 11101 10101.	I	..XX			xxgenpcvdm	v3.1			928	VSX Vector Generate PCV from Doubleword Mask X-form
111100 .... 11100 10101.	I	..XX			xxgenpcvhm	v3.1			931	VSX Vector Generate PCV from Halfword Mask X-form
111100 .... 11101 10100.	I	..XX			xxgenpcvwm	v3.1			933	VSX Vector Generate PCV from Word Mask X-form
111100 ..... /.... ..... 01011 0101..	I	..XX			xxinsertw	v3.0			917	VSX Vector Insert Word XX2-form
111100 .... 10000 010...	I	..XX			xxland	v2.06			907	VSX Vector Logical AND XX3-form
111100 .... 10001 010...	I	..XX			xxlandc	v2.06			907	VSX Vector Logical AND with Complement XX3-form
111100 .... 10111 010...	I	..XX			xxleqv	v2.07			907	VSX Vector Logical Equivalence XX3-form
111100 .... 10110 010...	I	..XX			xxlnand	v2.07			907	VSX Vector Logical NAND XX3-form
111100 .... 10100 010...	I	..XX			xxlnor	v2.06			908	VSX Vector Logical NOR XX3-form
111100 .... 10010 010...	I	..XX			xxlor	v2.06			908	VSX Vector Logical OR XX3-form
111100 .... 10101 010...	I	..XX			xxlorc	v2.07			908	VSX Vector Logical OR with Complement XX3-form
111100 .... 10011 010...	I	..XX			xxlxor	v2.06			908	VSX Vector Logical XOR XX3-form
111100 .... 00010 010...	I	..XX			xxmrghw	v2.06			918	VSX Vector Merge High Word XX3-form
111100 .... 00110 010...	I	..XX			xxmrglw	v2.06			918	VSX Vector Merge Low Word XX3-form
111100 .... 00011 010...	I	..XX			xxperm	v3.0			923	VSX Vector Permute XX3-form
111100 .... 0..01 010...	I	..XX			xxpermdi	v2.06			922	VSX Vector Permute Doubleword Immediate XX3-form
111100 .... 00111 010...	I	..XX			xxpermr	v3.0			923	VSX Vector Permute Right-indexed XX3-form

Table G.1: Power ISA Instruction Set Sorted by Compliancy Subset (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
000001 01000 0//// ///// ///// ///... 100010 .... 00....	I	..XX			xxpermx	v3.1			924	VSX Vector Permute Extended 8RR:XX4-form
111100 .... 11....	I	..XX			xxsel	v2.06			909	VSX Vector Select XX4-form
111100 .... 0..00 010...	I	..XX			xxsldwi	v2.06			925	VSX Vector Shift Left Double by Word Immediate XX3-form
000001 01000 0//// .... 100000 ..... 000..	I	..XX			xxsplti32dx	v3.1			919	VSX Vector Splat Immediate32 Doubleword Indexed 8RR:D-form
111100 ..... 00... ..... 01011 01000.	I	..XX			xxspltib	v3.0			919	VSX Vector Splat Immediate Byte X-form
000001 01000 0//// .... 100000 ..... 0010.	I	..XX			xxspltidp	v3.1			920	VSX Vector Splat Immediate Double-Precision 8RR:D-form
000001 01000 0//// .... 100000 ..... 0011.	I	..XX			xxspltiw	v3.1			920	VSX Vector Splat Immediate Word 8RR:D-form
111100 ..... ///.. ..... 01010 0100..	I	..XX			xxspltw	v2.06			921	VSX Vector Splat Word XX2-form
011111 ///// ///// ///// 01101 01110/	I	...X	BHRB		clrbhrb	v2.07			1041	Clear BHRB X-form
111011 .... 00000 00010.	I	...X	DFP		dadd[.]	v2.05			204	DFP Add X-form
111111 .... 00000 00010.	I	...X	DFP		daddq[.]	v2.05			204	DFP Add Quad X-form
111011 ..... ///// ..... 11001 00010.	I	...X	DFP		dcffix[.]	v2.06			227	DFP Convert From Fixed X-form
111111 ..... ///// ..... 11001 00010.	I	...X	DFP		dcffixq[.]	v2.05			227	DFP Convert From Fixed Quad X-form
111111 ..... 00000 ..... 11111 00010/	I	...X	DFP		dcffixqq	v3.1			227	DFP Convert From Fixed Quadword Quad X-form
111011 ...// 00100 00010/	I	...X	DFP		dcmpo	v2.05			209	DFP Compare Ordered X-form
111111 ...// 00100 00010/	I	...X	DFP		dcmpoq	v2.05			209	DFP Compare Ordered Quad X-form
111011 ...// 10100 00010/	I	...X	DFP		dcmpu	v2.05			208	DFP Compare Unordered X-form
111111 ...// 10100 00010/	I	...X	DFP		dcmpuq	v2.05			208	DFP Compare Unordered Quad X-form
111011 ..... ///// ..... 01000 00010.	I	...X	DFP		dctdp[.]	v2.05			225	DFP Convert To DFP Long X-form
111011 ..... ///// ..... 01001 00010.	I	...X	DFP		dctfix[.]	v2.05			228	DFP Convert To Fixed X-form
111111 ..... ///// ..... 01001 00010.	I	...X	DFP		dctfixq[.]	v2.05			228	DFP Convert To Fixed Quad X-form
111111 ..... 00001 ..... 11111 00010/	I	...X	DFP		dctfixqq	v3.1			228	DFP Convert To Fixed Quadword Quad X-form
111111 ..... ///// ..... 01000 00010.	I	...X	DFP		dctqpq[.]	v2.05			225	DFP Convert To DFP Extended X-form
111011 ....//// ..... 01010 00010.	I	...X	DFP		ddedpd[.]	v2.05			230	DFP Decode DPD To BCD X-form
111111 ....//// ..... 01010 00010.	I	...X	DFP		ddedpdq[.]	v2.05			230	DFP Decode DPD To BCD Quad X-form
111011 .... 10001 00010.	I	...X	DFP		ddiv[.]	v2.05			207	DFP Divide X-form
111111 .... 10001 00010.	I	...X	DFP		ddivq[.]	v2.05			207	DFP Divide Quad X-form
111011 ..... .//// ..... 11010 00010.	I	...X	DFP		denbcd[.]	v2.05			230	DFP Encode BCD To DPD X-form
111111 ..... .//// ..... 11010 00010.	I	...X	DFP		denbcdq[.]	v2.05			230	DFP Encode BCD To DPD Quad X-form
111011 .... 11011 00010.	I	...X	DFP		diex[.]	v2.05			231	DFP Insert Biased Exponent X-form
111111 .... 11011 00010.	I	...X	DFP		diexq[.]	v2.05			231	DFP Insert Biased Exponent Quad X-form
111011 .... 00001 00010.	I	...X	DFP		dmul[.]	v2.05			206	DFP Multiply X-form
111111 .... 00001 00010.	I	...X	DFP		dmulq[.]	v2.05			206	DFP Multiply Quad X-form
111011 ....000 00011.	I	...X	DFP		dqua[.]	v2.05			215	DFP Quantize Z23-form
111011 ....010 00011.	I	...X	DFP		dquai[.]	v2.05			214	DFP Quantize Immediate Z23-form
111111 ....010 00011.	I	...X	DFP		dquaiq[.]	v2.05			214	DFP Quantize Immediate Quad Z23-form
111111 ....000 00011.	I	...X	DFP		dquaq[.]	v2.05			215	DFP Quantize Quad Z23-form
111111 ..... ///// ..... 11000 00010.	I	...X	DFP		drdpq[.]	v2.05			226	DFP Round To DFP Long X-form
111011 ..... ////. ....111 00011.	I	...X	DFP		drintn[.]	v2.05			222	DFP Round To FP Integer Without Inexact Z23-form
111111 ..... ////. ....111 00011.	I	...X	DFP		drintnq[.]	v2.05			222	DFP Round To FP Integer Without Inexact Quad Z23-form
111011 ..... ////. ....011 00011.	I	...X	DFP		drintx[.]	v2.05			220	DFP Round To FP Integer With Inexact Z23-form
111111 ..... ////. ....011 00011.	I	...X	DFP		drintxq[.]	v2.05			220	DFP Round To FP Integer With Inexact Quad Z23-form
111011 ....001 00011.	I	...X	DFP		drrnd[.]	v2.05			217	DFP Reround Z23-form

Table G.1: Power ISA Instruction Set Sorted by Compliancy Subset (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
111111 ....001 00011.	I	...X	DFP		drrndq[.]	v2.05			217	DFP Reround Quad Z23-form
111011 ..... ///// ..... 11000 00010.	I	...X	DFP		drsp[.]	v2.05			226	DFP Round To DFP Short X-form
111011 .... .0010 00010.	I	...X	DFP		dscli[.]	v2.05			233	DFP Shift Significand Left Immediate Z22-form
111111 .... .0010 00010.	I	...X	DFP		dscliq[.]	v2.05			233	DFP Shift Significand Left Immediate Quad Z22-form
111011 .... .0011 00010.	I	...X	DFP		dscri[.]	v2.05			233	DFP Shift Significand Right Immediate Z22-form
111111 .... .0011 00010.	I	...X	DFP		dscriq[.]	v2.05			233	DFP Shift Significand Right Immediate Quad Z22-form
111011 .... 10000 00010.	I	...X	DFP		dsub[.]	v2.05			205	DFP Subtract X-form
111111 .... 10000 00010.	I	...X	DFP		dsubq[.]	v2.05			205	DFP Subtract Quad X-form
111011 ...// .... .0110 00010/	I	...X	DFP		dtstdc	v2.05			210	DFP Test Data Class Z22-form
111111 ...// .... .0110 00010/	I	...X	DFP		dtstdcq	v2.05			210	DFP Test Data Class Quad Z22-form
111011 ...// .... .0111 00010/	I	...X	DFP		dtstdg	v2.05			210	DFP Test Data Group Z22-form
111111 ...// .... .0111 00010/	I	...X	DFP		dtstdgq	v2.05			210	DFP Test Data Group Quad Z22-form
111011 ...// 00101 00010/	I	...X	DFP		dtstex	v2.05			211	DFP Test Exponent X-form
111111 ...// 00101 00010/	I	...X	DFP		dtstexq	v2.05			211	DFP Test Exponent Quad X-form
111011 ...// 10101 00010/	I	...X	DFP		dtstsf	v2.05			212	DFP Test Significance X-form
111011 ...// 10101 00011/	I	...X	DFP		dtstsfi	v3.0			213	DFP Test Significance Immediate X-form
111111 ...// 10101 00011/	I	...X	DFP		dtstsfiq	v3.0			213	DFP Test Significance Immediate Quad X-form
111111 ...// 10101 00010/	I	...X	DFP		dtstsfq	v2.05			212	DFP Test Significance Quad X-form
111011 ..... ///// ..... 01011 00010.	I	...X	DFP		dxex[.]	v2.05			231	DFP Extract Biased Exponent X-form
111111 ..... ///// ..... 01011 00010.	I	...X	DFP		dxexq[.]	v2.05			231	DFP Extract Biased Exponent Quad X-form
011111 .... 10011 00110/	II	...X	AMO		ldat	v3.0			1011	Load Doubleword Atomic X-form
111001 00	I	...X			lfdp	v2.05			158	Load Floating-Point Double Pair DS-form
011111 .... 11000 10111/	I	...X			lfdpx	v2.05			158	Load Floating-Point Double Pair Indexed X-form
101110	I	...X			lmw	P1			68	Load Multiple Word D-form
011111 .... 10010 10101/	I	...X			lswi	P1			70	Load String Word Immediate X-form
011111 .... 10000 10101/	I	...X			lswx	P1			70	Load String Word Indexed X-form
011111 .... 10010 00110/	II	...X	AMO		lwat	v3.0			1011	Load Word Atomic X-form
011111 .... 01001 01110/	I	...X	BHRB		mfbhrbe	v2.07			1041	Move From Branch History Rolling Buffer Entry XFX-form
010011 ///// ///// ////. 00100 10010/	I	...X	EBB		rfebb	v2.07			1037	Return from Event-Based Branch XL-form
011111 ..... ///// 11110 100111	III	...X			slbfee.	v2.05	P	SR	1170	SLB Find Entry ESID X-form
011111 ..... ////. ///// 11010 10010/	III	...X			slbiag	v3.0B	P		1167	SLB Invalidate All Global X-form
011111 ///// ///// 01101 10010/	III	...X			slbie	PPC	P		1160	SLB Invalidate Entry X-form
011111 ..... ///// 01110 10010/	III	...X			slbieg	v3.0	P		1162	SLB Invalidate Entry Global X-form
011111 ..... ///// 11100 10011/	III	...X			slbmfee	v2.00	P		1169	SLB Move From Entry ESID X-form
011111 ..... ///// 11010 10011/	III	...X			slbmfev	v2.00	P		1169	SLB Move From Entry VSID X-form
011111 ..... ///// 01100 10010/	III	...X			slbmte	v2.00	P		1168	SLB Move To Entry X-form
011111 ///// ///// ///// 01010 10010/	III	...X			slbsync	v3.0	P		1171	SLB Synchronize X-form
011111 .... 10111 00110/	II	...X	AMO		stdat	v3.0			1013	Store Doubleword Atomic X-form
111101 00	I	...X			stfdp	v2.05			159	Store Floating-Point Double Pair DS-form
011111 .... 11100 10111/	I	...X			stfdpx	v2.05			159	Store Floating-Point Double Pair Indexed X-form
101111	I	...X			stmw	P1			68	Store Multiple Word D-form
011111 .... 10110 10101/	I	...X			stswi	P1			71	Store String Word Immediate X-form
011111 .... 10100 10101/	I	...X			stswx	P1			71	Store String Word Indexed X-form
011111 .... 10110 00110/	II	...X	AMO		stwat	v3.0			1013	Store Word Atomic X-form
011111 ..... /.... ..... 01001 10010/	III	...X			tlbie	P1	HV	64	1174	TLB Invalidate Entry X-form
011111 ///// ///// ///// 10001 10110/	III	...X			tlbsync	PPC	HV/P		1186	TLB Synchronize X-form
111111 ..... 00000 ..... 11001 00100/	I	...X	BFP128		xsabsqp	v3.0			607	VSX Scalar Absolute Quad-Precision X-form

Table G.1: Power ISA Instruction Set Sorted by Compliancy Subset (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
111111 .... 00000 00100.	I	...X	BFP128		xsaddqp[o]	v3.0			620	VSX Scalar Add Quad-Precision [using round to Odd] X-form
111111 ...// 00101 00100/	I	...X	BFP128		xscmpexpqp	v3.0			863	VSX Scalar Compare Exponents Quad-Precision X-form
111111 ...// 00100 00100/	I	...X	BFP128		xscmpoqp	v3.0			745	VSX Scalar Compare Ordered Quad-Precision X-form
111111 ...// 10100 00100/	I	...X	BFP128		xscmpuqp	v3.0			748	VSX Scalar Compare Unordered Quad-Precision X-form
111111 .... 00011 00100/	I	...X	BFP128		xscpsgnqp	v3.0			608	VSX Scalar Copy Sign Quad-Precision X-form
111111 ..... 10110 ..... 11010 00100/	I	...X	BFP128		xscvdpqp	v3.0			795	VSX Scalar Convert Double-Precision to Quad-Precision format X-form
111111 ..... 10100 ..... 11010 00100.	I	...X	BFP128		xscvqpdp[o]	v3.0			791	VSX Scalar Convert with round Quad-Precision to Double-Precision format [using round to Odd] X-form
111111 ..... 11001 ..... 11010 00100/	I	...X	BFP128		xscvqpsdz	v3.0			824	VSX Scalar Convert with round to zero Quad-Precision to Signed Doubleword format X-form
111111 ..... 01001 ..... 11010 00100/	I	...X	BFP128		xscvqpswz	v3.0			828	VSX Scalar Convert with round to zero Quad-Precision to Signed Word format X-form
111111 ..... 10001 ..... 11010 00100/	I	...X	BFP128		xscvqpudz	v3.0			830	VSX Scalar Convert with round to zero Quad-Precision to Unsigned Doubleword format X-form
111111 ..... 00001 ..... 11010 00100/	I	...X	BFP128		xscvqpuwz	v3.0			834	VSX Scalar Convert with round to zero Quad-Precision to Unsigned Word format X-form
111111 ..... 01010 ..... 11010 00100/	I	...X	BFP128		xscvsdqp	v3.0			852	VSX Scalar Convert Signed Doubleword to Quad-Precision format X-form
111111 ..... 00010 ..... 11010 00100/	I	...X	BFP128		xscvudqp	v3.0			852	VSX Scalar Convert Unsigned Doubleword to Quad-Precision format X-form
111111 .... 10001 00100.	I	...X	BFP128		xsdivqp[o]	v3.0			626	VSX Scalar Divide Quad-Precision [using round to Odd] X-form
111111 .... 11011 00100/	I	...X	BFP128		xsiexpqp	v3.0			865	VSX Scalar Insert Exponent Quad-Precision X-form
111111 .... 01100 00100.	I	...X	BFP128		xsmaddqp[o]	v3.0			654	VSX Scalar Multiply-Add Quad-Precision [using round to Odd] X-form
111111 .... 01101 00100.	I	...X	BFP128		xsmsubqp[o]	v3.0			663	VSX Scalar Multiply-Subtract Quad-Precision [using round to Odd] X-form
111111 .... 00001 00100.	I	...X	BFP128		xsmulqp[o]	v3.0			632	VSX Scalar Multiply Quad-Precision [using round to Odd] X-form
111111 ..... 01000 ..... 11001 00100/	I	...X	BFP128		xsnabsqp	v3.0			609	VSX Scalar Negative Absolute Quad-Precision X-form
111111 ..... 10000 ..... 11001 00100/	I	...X	BFP128		xsnegqp	v3.0			610	VSX Scalar Negate Quad-Precision X-form
111111 .... 01110 00100.	I	...X	BFP128		xsnmaddqp[o]	v3.0			673	VSX Scalar Negative Multiply-Add Quad-Precision [using round to Odd] X-form
111111 .... 01111 00100.	I	...X	BFP128		xsnmsubqp[o]	v3.0			682	VSX Scalar Negative Multiply-Subtract Quad-Precision [using round to Odd] X-form
111111 ..... ////. ....000 00101.	I	...X	BFP128		xsrqpi[x]	v3.0			806	VSX Scalar Round to Quad-Precision Integer [with Inexact] Z23-form
111111 ..... ////. 001 00101/	I	...X	BFP128		xsrqpxp	v3.0			785	VSX Scalar Round Quad-Precision to Double-Extended-Precision Z23-form
111111 ..... 11011 ..... 11001 00100.	I	...X	BFP128		xssqrtqp[o]	v3.0			638	VSX Scalar Square Root Quad-Precision [using round to Odd] X-form
111111 .... 10000 00100.	I	...X	BFP128		xssubqp[o]	v3.0			644	VSX Scalar Subtract Quad-Precision [using round to Odd] X-form
111111 .... 10110 00100/	I	...X	BFP128		xststdcqp	v3.0			867	VSX Scalar Test Data Class Quad-Precision X-form
111111 ..... 00010 ..... 11001 00100/	I	...X	BFP128		xsxexpqp	v3.0			869	VSX Scalar Extract Exponent Quad-Precision X-form
111111 ..... 10010 ..... 11001 00100/	I	...X	BFP128		xsxsigqp	v3.0			870	VSX Scalar Extract Significand Quad-Precision X-form
011111 ////1 .... 11000 00110/	II	....			copy	v3.0			1007	Copy X-form

Table G.1: Power ISA Instruction Set Sorted by Compliancy Subset (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
011111 ///// ///// ///// 11010 00110/	II	....			cpabort	v3.0			1008	Copy-Paste Abort X-form
011111 .... 10111 10010.	I	....			hashchk	v3.1B			120	Hash Check X-form
011111 .... 10101 10010.	III	....			hashchkp	v3.1B	P		1099	Hash Check Privileged X-form
011111 .... 10110 10010.	I	....			hashst	v3.1B			120	Hash Store X-form
011111 .... 10100 10010.	III	....			hashstp	v3.1B	P		1099	Hash Store Privileged X-form
011111 ///// ///// 00011 01110/	III	....			msgclru	v3.0C	UV		1274	Message Clear Ultravisor X-form
011111 ///// ///// 00010 01110/	III	....			msgsndu	v3.0C	UV		1274	Message Send Ultravisor X-form
011111 ////. .... 11100 00110.	II	....			paste.	v3.0			1008	Paste X-form
000001 11100 1//// ..//// ///.. ..... 111011 .../// .... 00110 011../	I	....	MMA	MMA	pmxvbf16ger2	v3.1			889	Prefixed Masked VSX Vector bfloat16 GER (rank-2 update) MMIRR:XX3-form
000001 11100 1//// ..//// ///.. ..... 111011 .../// .... 11110 010../	I	....	MMA	MMA	pmxvbf16ger2nn	v3.1			890	Prefixed Masked VSX Vector bfloat16 GER (rank-2 update) Negative multiply, Negative accumulate MMIRR:XX3-form
000001 11100 1//// ..//// ///.. ..... 111011 .../// .... 01110 010../	I	....	MMA	MMA	pmxvbf16ger2np	v3.1			890	Prefixed Masked VSX Vector bfloat16 GER (rank-2 update) Negative multiply, Positive accumulate MMIRR:XX3-form
000001 11100 1//// ..//// ///.. ..... 111011 .../// .... 10110 010../	I	....	MMA	MMA	pmxvbf16ger2pn	v3.1			889	Prefixed Masked VSX Vector bfloat16 GER (rank-2 update) Positive multiply, Negative accumulate MMIRR:XX3-form
000001 11100 1//// ..//// ///.. ..... 111011 .../// .... 00110 010../	I	....	MMA	MMA	pmxvbf16ger2pp	v3.1			889	Prefixed Masked VSX Vector bfloat16 GER (rank-2 update) Positive multiply, Positive accumulate MMIRR:XX3-form
000001 11100 1//// ..//// ///.. ..... 111011 .../// .... 00010 011../	I	....	MMA	MMA	pmxvf16ger2	v3.1			894	Prefixed Masked VSX Vector 16-bit Floating-Point GER (rank-2 update) MMIRR:XX3-form
000001 11100 1//// ..//// ///.. ..... 111011 .../// .... 11010 010../	I	....	MMA	MMA	pmxvf16ger2nn	v3.1			895	Prefixed Masked VSX Vector 16-bit Floating-Point GER (rank-2 update) Negative multiply, Negative accumulate MMIRR:XX3-form
000001 11100 1//// ..//// ///.. ..... 111011 .../// .... 01010 010../	I	....	MMA	MMA	pmxvf16ger2np	v3.1			895	Prefixed Masked VSX Vector 16-bit Floating-Point GER (rank-2 update) Negative multiply, Positive accumulate MMIRR:XX3-form
000001 11100 1//// ..//// ///.. ..... 111011 .../// .... 10010 010../	I	....	MMA	MMA	pmxvf16ger2pn	v3.1			894	Prefixed Masked VSX Vector 16-bit Floating-Point GER (rank-2 update) Positive multiply, Negative accumulate MMIRR:XX3-form
000001 11100 1//// ..//// ///.. ..... 111011 .../// .... 00010 010../	I	....	MMA	MMA	pmxvf16ger2pp	v3.1			894	Prefixed Masked VSX Vector 16-bit Floating-Point GER (rank-2 update) Positive multiply, Positive accumulate MMIRR:XX3-form
000001 11100 1//// ///// ///.. ..... 111011 .../// .... 00011 011../	I	....	MMA	MMA	pmxvf32ger	v3.1			899	Prefixed Masked VSX Vector 32-bit Floating-Point GER (rank-1 update) MMIRR:XX3-form
000001 11100 1//// ///// ///.. ..... 111011 .../// .... 11011 010../	I	....	MMA	MMA	pmxvf32gernn	v3.1			900	Prefixed Masked VSX Vector 32-bit Floating-Point GER (rank-1 update) Negative multiply, Negative accumulate MMIRR:XX3-form
000001 11100 1//// ///// ///.. ..... 111011 .../// .... 01011 010../	I	....	MMA	MMA	pmxvf32gernp	v3.1			900	Prefixed Masked VSX Vector 32-bit Floating-Point GER (rank-1 update) Negative multiply, Positive accumulate MMIRR:XX3-form
000001 11100 1//// ///// ///.. ..... 111011 .../// .... 10011 010../	I	....	MMA	MMA	pmxvf32gerpn	v3.1			899	Prefixed Masked VSX Vector 32-bit Floating-Point GER (rank-1 update) Positive multiply, Negative accumulate MMIRR:XX3-form

Table G.1: Power ISA Instruction Set Sorted by Compliancy Subset (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
000001 11100 1//// ///// ///.. ..... 111011 ...// .... 00011 010../	I	....	MMA	MMA	pmxvf32gerpp	v3.1			899	Prefixed Masked VSX Vector 32-bit Floating-Point GER (rank-1 update) Positive multiply, Positive accumulate MMIRR:XX3-form
000001 11100 1//// ///// ///.. ..... 111011 ...// .... 00111 011../	I	....	MMA	MMA	pmxvf64ger	v3.1			903	Prefixed Masked VSX Vector 64-bit Floating-Point GER (rank-1 update) MMIRR:XX3-form
000001 11100 1//// ///// ///.. ..... 111011 ...// .... 11111 010../	I	....	MMA	MMA	pmxvf64gernn	v3.1			904	Prefixed Masked VSX Vector 64-bit Floating-Point GER (rank-1 update) Negative multiply, Negative accumulate MMIRR:XX3-form
000001 11100 1//// ///// ///.. ..... 111011 ...// .... 01111 010../	I	....	MMA	MMA	pmxvf64gernp	v3.1			904	Prefixed Masked VSX Vector 64-bit Floating-Point GER (rank-1 update) Negative multiply, Positive accumulate MMIRR:XX3-form
000001 11100 1//// ///// ///.. ..... 111011 ...// .... 10111 010../	I	....	MMA	MMA	pmxvf64gerpn	v3.1			903	Prefixed Masked VSX Vector 64-bit Floating-Point GER (rank-1 update) Positive multiply, Negative accumulate MMIRR:XX3-form
000001 11100 1//// ///// ///.. ..... 111011 ...// .... 00111 010../	I	....	MMA	MMA	pmxvf64gerpp	v3.1			903	Prefixed Masked VSX Vector 64-bit Floating-Point GER (rank-1 update) Positive multiply, Positive accumulate MMIRR:XX3-form
000001 11100 1//// ..//// ///.. ..... 111011 ...// .... 01001 011../	I	....	MMA	MMA	pmxvi16ger2	v3.1			878	Prefixed Masked VSX Vector 16-bit Signed Integer GER (rank-2 update) MMIRR:XX3-form
000001 11100 1//// ..//// ///.. ..... 111011 ...// .... 01101 011../	I	....	MMA	MMA	pmxvi16ger2pp	v3.1			878	Prefixed Masked VSX Vector 16-bit Signed Integer GER (rank-2 update) Positive multiply, Positive accumulate MMIRR:XX3-form
000001 11100 1//// ..//// ///.. ..... 111011 ...// .... 00101 011../	I	....	MMA	MMA	pmxvi16ger2s	v3.1			880	Prefixed Masked VSX Vector 16-bit Signed Integer GER (rank-2 update) with Saturation MMIRR:XX3-form
000001 11100 1//// ..//// ///.. ..... 111011 ...// .... 00101 010../	I	....	MMA	MMA	pmxvi16ger2spp	v3.1			880	Prefixed Masked VSX Vector 16-bit Signed Integer GER (rank-2 update) with Saturation Positive multiply, Positive accumulate MMIRR:XX3-form
000001 11100 1//// .... 111011 ...// .... 00100 011../	I	....	MMA	MMA	pmxvi4ger8	v3.1			882	Prefixed Masked VSX Vector 4-bit Signed Integer GER (rank-8 update) MMIRR:XX3-form
000001 11100 1//// .... 111011 ...// .... 00100 010../	I	....	MMA	MMA	pmxvi4ger8pp	v3.1			882	Prefixed Masked VSX Vector 4-bit Signed Integer GER (rank-8 update) Positive multiply, Positive accumulate MMIRR:XX3-form
000001 11100 1//// ....// ///.. ..... 111011 ...// .... 00000 011../	I	....	MMA	MMA	pmxvi8ger4	v3.1			885	Prefixed Masked VSX Vector 8-bit Signed/Unsigned Integer GER (rank-4 update) MMIRR:XX3-form
000001 11100 1//// ....// ///.. ..... 111011 ...// .... 00000 010../	I	....	MMA	MMA	pmxvi8ger4pp	v3.1			885	Prefixed Masked VSX Vector 8-bit Signed/Unsigned Integer GER (rank-4 update) Positive multiply, Positive accumulate MMIRR:XX3-form
000001 11100 1//// ....// ///.. ..... 111011 ...// .... 01100 011../	I	....	MMA	MMA	pmxvi8ger4spp	v3.1			887	Prefixed Masked VSX Vector 8-bit Signed/Unsigned Integer GER (rank-4 update) with Saturation Positive multiply, Positive accumulate MMIRR:XX3-form
010011 ///// ///// ///// 01001 10010/	III	....			urfid	v3.0C	UV		1087	Ultravisor Return From Interrupt Doubleword XL-form
111011 ...// .... 00110 011../	I	....	MMA	MMA	xvbf16ger2	v3.1			889	VSX Vector bfloat16 GER (rank-2 update) XX3-form
111011 ...// .... 11110 010../	I	....	MMA	MMA	xvbf16ger2nn	v3.1			889	VSX Vector bfloat16 GER (rank-2 update) Negative multiply, Negative accumulate XX3-form
111011 ...// .... 01110 010../	I	....	MMA	MMA	xvbf16ger2np	v3.1			889	VSX Vector bfloat16 GER (rank-2 update) Negative multiply, Positive accumulate XX3-form
111011 ...// .... 10110 010../	I	....	MMA	MMA	xvbf16ger2pn	v3.1			889	VSX Vector bfloat16 GER (rank-2 update) Positive multiply, Negative accumulate XX3-form

Table G.1: Power ISA Instruction Set Sorted by Compliancy Subset (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
111011 ...// .... 00110 010../	I	....	MMA	MMA	xvbf16ger2pp	v3.1			889	VSX Vector bfloat16 GER (rank-2 update) Positive multiply, Positive accumulate XX3-form
111011 ...// .... 00010 011../	I	....	MMA	MMA	xvf16ger2	v3.1			894	VSX Vector 16-bit Floating-Point GER (rank-2 update) XX3-form
111011 ...// .... 11010 010../	I	....	MMA	MMA	xvf16ger2nn	v3.1			894	VSX Vector 16-bit Floating-Point GER (rank-2 update) Negative multiply, Negative accumulate XX3-form
111011 ...// .... 01010 010../	I	....	MMA	MMA	xvf16ger2np	v3.1			894	VSX Vector 16-bit Floating-Point GER (rank-2 update) Negative multiply, Positive accumulate XX3-form
111011 ...// .... 10010 010../	I	....	MMA	MMA	xvf16ger2pn	v3.1			894	VSX Vector 16-bit Floating-Point GER (rank-2 update) Positive multiply, Negative accumulate XX3-form
111011 ...// .... 00010 010../	I	....	MMA	MMA	xvf16ger2pp	v3.1			894	VSX Vector 16-bit Floating-Point GER (rank-2 update) Positive multiply, Positive accumulate XX3-form
111011 ...// .... 00011 011../	I	....	MMA	MMA	xvf32ger	v3.1			899	VSX Vector 32-bit Floating-Point GER (rank-1 update) XX3-form
111011 ...// .... 11011 010../	I	....	MMA	MMA	xvf32gernn	v3.1			899	VSX Vector 32-bit Floating-Point GER (rank-1 update) Negative multiply, Negative accumulate XX3-form
111011 ...// .... 01011 010../	I	....	MMA	MMA	xvf32gernp	v3.1			899	VSX Vector 32-bit Floating-Point GER (rank-1 update) Negative multiply, Positive accumulate XX3-form
111011 ...// .... 10011 010../	I	....	MMA	MMA	xvf32gerpn	v3.1			899	VSX Vector 32-bit Floating-Point GER (rank-1 update) Positive multiply, Negative accumulate XX3-form
111011 ...// .... 00011 010../	I	....	MMA	MMA	xvf32gerpp	v3.1			899	VSX Vector 32-bit Floating-Point GER (rank-1 update) Positive multiply, Positive accumulate XX3-form
111011 ...// .... 00111 011../	I	....	MMA	MMA	xvf64ger	v3.1			903	VSX Vector 64-bit Floating-Point GER (rank-1 update) XX3-form
111011 ...// .... 11111 010../	I	....	MMA	MMA	xvf64gernn	v3.1			903	VSX Vector 64-bit Floating-Point GER (rank-1 update) Negative multiply, Negative accumulate XX3-form
111011 ...// .... 01111 010../	I	....	MMA	MMA	xvf64gernp	v3.1			903	VSX Vector 64-bit Floating-Point GER (rank-1 update) Negative multiply, Positive accumulate XX3-form
111011 ...// .... 10111 010../	I	....	MMA	MMA	xvf64gerpn	v3.1			903	VSX Vector 64-bit Floating-Point GER (rank-1 update) Positive multiply, Negative accumulate XX3-form
111011 ...// .... 00111 010../	I	....	MMA	MMA	xvf64gerpp	v3.1			903	VSX Vector 64-bit Floating-Point GER (rank-1 update) Positive multiply, Positive accumulate XX3-form
111011 ...// .... 01001 011../	I	....	MMA	MMA	xvi16ger2	v3.1			878	VSX Vector 16-bit Signed Integer GER (rank-2 update) XX3-form
111011 ...// .... 01101 011../	I	....	MMA	MMA	xvi16ger2pp	v3.1			878	VSX Vector 16-bit Signed Integer GER (rank-2 update) Positive multiply, Positive accumulate XX3-form
111011 ...// .... 00101 011../	I	....	MMA	MMA	xvi16ger2s	v3.1			880	VSX Vector 16-bit Signed Integer GER (rank-2 update) with Saturation XX3-form
111011 ...// .... 00101 010../	I	....	MMA	MMA	xvi16ger2spp	v3.1			880	VSX Vector 16-bit Signed Integer GER (rank-2 update) with Saturation Positive multiply, Positive accumulate XX3-form
111011 ...// .... 00100 011../	I	....	MMA	MMA	xvi4ger8	v3.1			882	VSX Vector 4-bit Signed Integer GER (rank-8 update) XX3-form
111011 ...// .... 00100 010../	I	....	MMA	MMA	xvi4ger8pp	v3.1			882	VSX Vector 4-bit Signed Integer GER (rank-8 update) Positive multiply, Positive accumulate XX3-form
111011 ...// .... 00000 011../	I	....	MMA	MMA	xvi8ger4	v3.1			885	VSX Vector 8-bit Signed/Unsigned Integer GER (rank-4 update) XX3-form
111011 ...// .... 00000 010../	I	....	MMA	MMA	xvi8ger4pp	v3.1			885	VSX Vector 8-bit Signed/Unsigned Integer GER (rank-4 update) Positive multiply, Positive accumulate XX3-form

Table G.1: Power ISA Instruction Set Sorted by Compliancy Subset (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
111011 …// .... 01100 011../	I	....	MMA	MMA	xvi8ger4spp	v3.1			887	VSX Vector 8-bit Signed/Unsigned Integer GER (rank-4 update) with Saturation Positive multiply, Positive accumulate XX3-form
011111 …// 00000 ///// 00101 10001/	I	....	MMA	MMA	xxmfacc	v3.1			876	VSX Move From Accumulator X-form
011111 …// 00001 ///// 00101 10001/	I	....	MMA	MMA	xxmtacc	v3.1			877	VSX Move To Accumulator X-form
011111 …// 00011 ///// 00101 10001/	I	....	MMA	MMA	xxsetaccz	v3.1			877	VSX Set Accumulator to Zero X-form

Appendix H. Power ISA Instruction Set Sorted by Mnemonic

This appendix lists all the instructions in the Power ISA, sorted by mnemonic. For an explanation of the values in the columns, see Appendix E.

Table H.1: Power ISA Instruction Set Sorted by Mnemonic

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
011111 .... 01000 01010.	I	XXXX			add[.]	P1		SR	74	Add XO-form
011111 .... 00000 01010.	I	XXXX			addc[.]	P1		SR	76	Add Carrying XO-form
011111 .... 10000 01010.	I	..XX			addco[.]	P1		SR	76	Add Carrying & record OV XO-form
011111 .... 00100 01010.	I	XXXX			adde[.]	P1		SR	76	Add Extended XO-form
011111 .... 10100 01010.	I	..XX			addeo[.]	P1		SR	76	Add Extended & record OV XO-form
011111 ....101 01010/	I	XXXX			addex	v3.0B			78	Add Extended using alternate carry bit Z23-form
011111 .... /0010 01010/	I	XXXX			addg6s	v2.06			117	Add and Generate Sixes XO-form
001110	I	XXXX			addi	P1			73	Add Immediate D-form
001100	I	XXXX			addic	P1		SR	75	Add Immediate Carrying D-form
001101	I	XXXX			addic.	P1		SR	75	Add Immediate Carrying and Record D-form
001111	I	XXXX			addis	P1			74	Add Immediate Shifted D-form
011111 .... ///// 00111 01010.	I	XXXX			addme[.]	P1		SR	77	Add to Minus One Extended XO-form
011111 .... ///// 10111 01010.	I	..XX			addmeo[.]	P1		SR	77	Add to Minus One Extended & record OV XO-form
011111 .... 11000 01010.	I	..XX			addo[.]	P1		SR	74	Add & record OV XO-form
010011 .... 00010.	I	XXXX			addpcis	v3.0			74	Add PC Immediate Shifted DX-form
011111 .... ///// 00110 01010.	I	XXXX			addze[.]	P1		SR	77	Add to Zero Extended XO-form
011111 .... ///// 10110 01010.	I	..XX			addzeo[.]	P1		SR	77	Add to Zero Extended & record OV XO-form
011111 .... 00000 11100.	I	XXXX			and[.]	P1		SR	99	AND X-form
011111 .... 00001 11100.	I	XXXX			andc[.]	P1		SR	100	AND with Complement X-form
011100	I	XXXX			andi.	P1		SR	97	AND Immediate D-form
011101	I	XXXX			andis.	P1		SR	97	AND Immediate Shifted D-form
010010	I	XXXX			b[l][a]	P1			40	Branch I-form
010000	I	XXXX			bc[l][a]	P1		CT	40	Branch Conditional B-form
010011 .... ///.. 10000 10000.	I	XXXX			bcctr[l]	P1		CT	41	Branch Conditional to Count Register XL-form
000100 .... 1.000 000001	I	..XX			bcdadd.	v2.07			466	Decimal Add Modulo VX-form
000100 ..... 00111 ..... 1.110 000001	I	..XX			bcdcfn.	v3.0			468	Decimal Convert From National VX-form
000100 ..... 00010 ..... 1.110 000001	I	..XX			bcdcfsq.	v3.0			472	Decimal Convert From Signed Quadword VX-form
000100 ..... 00110 ..... 1.110 000001	I	..XX			bcdcfz.	v3.0			469	Decimal Convert From Zoned VX-form
000100 .... 01101 000001	I	..XX			bcdcpsgn.	v3.0			476	Decimal Copy Sign VX-form
000100 ..... 00101 ..... 1/110 000001	I	..XX			bcdctn.	v3.0			470	Decimal Convert To National VX-form
000100 ..... 00000 ..... 1/110 000001	I	..XX			bcdctsq.	v3.0			473	Decimal Convert To Signed Quadword VX-form
000100 ..... 00100 ..... 1.110 000001	I	..XX			bcdctz.	v3.0			471	Decimal Convert To Zoned VX-form
000100 .... 1.011 000001	I	..XX			bcds.	v3.0			478	Decimal Shift VX-form

Table H.1: Power ISA Instruction Set Sorted by Mnemonic (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
000100 ..... 11111 ..... 1.110 000001	I	..XX			bcdsetsgn.	v3.0			477	Decimal Set Sign VX-form
000100 .... 1.111 000001	I	..XX			bcdsr.	v3.0			480	Decimal Shift & Round VX-form
000100 .... 1.001 000001	I	..XX			bcdsub.	v2.07			466	Decimal Subtract Modulo VX-form
000100 .... 1.100 000001	I	..XX			bcdtrunc.	v3.0			481	Decimal Truncate VX-form
000100 .... 1/010 000001	I	..XX			bcdus.	v3.0			479	Decimal Unsigned Shift VX-form
000100 .... 1/101 000001	I	..XX			bcdutrunc.	v3.0			482	Decimal Unsigned Truncate VX-form
010011 .... ///.. 00000 10000.	I	XXXX			bclr[l]	P1		CT	41	Branch Conditional to Link Register XL-form
010011 .... ///.. 10001 10000.	I	XXXX			bctar[l]	v2.07			42	Branch Conditional to Branch Target Address Register XL-form
011111 .... 00111 11100/	I	..XX			bpermd	v2.06			105	Bit Permute Doubleword X-form
011111 .... ///// 00101 11011/	I	..XX			brd	v3.1			118	Byte-Reverse Doubleword X-form
011111 .... ///// 00110 11011/	I	XXXX			brh	v3.1			118	Byte-Reverse Halfword X-form
011111 .... ///// 00100 11011/	I	XXXX			brw	v3.1			118	Byte-Reverse Word X-form
011111 .... ///// 01001 11010/	I	XXXX			cbcdtd	v2.06			117	Convert Binary Coded Decimal To Declets X-form
011111 .... ///// 01000 11010/	I	XXXX			cdtbcd	v2.06			117	Convert Declets To Binary Coded Decimal X-form
011111 .... 00110 11100/	I	..XX			cfuged	v3.1			105	Centrifuge Doubleword X-form
011111 ///// ///// ///// 01101 01110/	I	...X	BHRB		clrbhrb	v2.07			1041	Clear BHRB X-form
011111 .../. 00000 00000/	I	XXXX			cmp	P1			90	Compare X-form
011111 .... 01111 11100/	I	XXXX			cmpb	v2.05			101	Compare Bytes X-form
011111 .../// 00111 00000/	I	XXXX			cmpeqb	v3.0			93	Compare Equal Byte X-form
001011 .../.	I	XXXX			cmpi	P1			90	Compare Immediate D-form
011111 .../. 00001 00000/	I	XXXX			cmpl	P1			91	Compare Logical X-form
001010 .../.	I	XXXX			cmpli	P1			91	Compare Logical Immediate D-form
011111 .../. 00110 00000/	I	XXXX			cmprb	v3.0			92	Compare Ranged Byte X-form
011111 .... ///// 00001 11010.	I	..XX			cntlzd[.]	PPC		SR	104	Count Leading Zeros Doubleword X-form
011111 .... 00001 11011/	I	..XX			cntlzdm	v3.1			104	Count Leading Zeros Doubleword under bit Mask X-form
011111 .... ///// 00000 11010.	I	XXXX			cntlzw[.]	P1		SR	101	Count Leading Zeros Word X-form
011111 .... ///// 10001 11010.	I	..XX			cnttzd[.]	v3.0			104	Count Trailing Zeros Doubleword X-form
011111 .... 10001 11011/	I	..XX			cnttzdm	v3.1			104	Count Trailing Zeros Doubleword under bit Mask X-form
011111 .... ///// 10000 11010.	I	XXXX			cnttzw[.]	v3.0			101	Count Trailing Zeros Word X-form
011111 ///1 .... 11000 00110/	II	....			copy	v3.0			1007	Copy X-form
011111 ///// ///// ///// 11010 00110/	II	....			cpabort	v3.0			1008	Copy-Paste Abort X-form
010011 .... 01000 00001/	I	XXXX			crand	P1			43	Condition Register AND XL-form
010011 .... 00100 00001/	I	XXXX			crandc	P1			44	Condition Register AND with Complement XL-form
010011 .... 01001 00001/	I	XXXX			creqv	P1			44	Condition Register Equivalent XL-form
010011 .... 00111 00001/	I	XXXX			crnand	P1			43	Condition Register NAND XL-form
010011 .... 00001 00001/	I	XXXX			crnor	P1			44	Condition Register NOR XL-form
010011 .... 01110 00001/	I	XXXX			cror	P1			43	Condition Register OR XL-form
010011 .... 01101 00001/	I	XXXX			crorc	P1			44	Condition Register OR with Complement XL-form
010011 .... 00110 00001/	I	XXXX			crxor	P1			43	Condition Register XOR XL-form
111011 .... 00000 00010.	I	...X	DFP		dadd[.]	v2.05			204	DFP Add X-form
111111 .... 00000 00010.	I	...X	DFP		daddq[.]	v2.05			204	DFP Add Quad X-form
011111 ..... ///.. ///// 10111 10011/	I	XXXX			darn	v3.0			84	Deliver A Random Number X-form
011111 //... .... 00010 10110/	II	..XX			dcbf	PPC			1003	Data Cache Block Flush X-form
011111 ///// 00001 10110/	II	..XX			dcbst	PPC			1002	Data Cache Block Store X-form
011111 .... 01000 10110/	II	..XX			dcbt	PPC			999	Data Cache Block Touch X-form
011111 .... 00111 10110/	II	..XX			dcbtst	PPC			1000	Data Cache Block Touch for Store X-form
011111 ///// 11111 10110/	II	XXXX			dcbz	P1			1001	Data Cache Block set to Zero X-form

Table H.1: Power ISA Instruction Set Sorted by Mnemonic (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
111011 ..... ///// ..... 11001 00010.	I	...X	DFP		dcffix[.]	v2.06			227	DFP Convert From Fixed X-form
111111 ..... ///// ..... 11001 00010.	I	...X	DFP		dcffixq[.]	v2.05			227	DFP Convert From Fixed Quad X-form
111111 ..... 00000 ..... 11111 00010/	I	...X	DFP		dcffixqq	v3.1			227	DFP Convert From Fixed Quadword Quad X-form
111011 ...// 00100 00010/	I	...X	DFP		dcmpo	v2.05			209	DFP Compare Ordered X-form
111111 ...// 00100 00010/	I	...X	DFP		dcmpoq	v2.05			209	DFP Compare Ordered Quad X-form
111011 ...// 10100 00010/	I	...X	DFP		dcmpu	v2.05			208	DFP Compare Unordered X-form
111111 ...// 10100 00010/	I	...X	DFP		dcmpuq	v2.05			208	DFP Compare Unordered Quad X-form
111011 ..... ///// ..... 01000 00010.	I	...X	DFP		dctdp[.]	v2.05			225	DFP Convert To DFP Long X-form
111011 ..... ///// ..... 01001 00010.	I	...X	DFP		dctfix[.]	v2.05			228	DFP Convert To Fixed X-form
111111 ..... ///// ..... 01001 00010.	I	...X	DFP		dctfixq[.]	v2.05			228	DFP Convert To Fixed Quad X-form
111111 ..... 00001 ..... 11111 00010/	I	...X	DFP		dctfixqq	v3.1			228	DFP Convert To Fixed Quadword Quad X-form
111111 ..... ///// ..... 01000 00010.	I	...X	DFP		dctqpq[.]	v2.05			225	DFP Convert To DFP Extended X-form
111011 ....//// ..... 01010 00010.	I	...X	DFP		ddedpd[.]	v2.05			230	DFP Decode DPD To BCD X-form
111111 ....//// ..... 01010 00010.	I	...X	DFP		ddedpdq[.]	v2.05			230	DFP Decode DPD To BCD Quad X-form
111011 .... 10001 00010.	I	...X	DFP		ddiv[.]	v2.05			207	DFP Divide X-form
111111 .... 10001 00010.	I	...X	DFP		ddivq[.]	v2.05			207	DFP Divide Quad X-form
111011 ..... ./// ..... 11010 00010.	I	...X	DFP		denbcd[.]	v2.05			230	DFP Encode BCD To DPD X-form
111111 ..... .//// ..... 11010 00010.	I	...X	DFP		denbcdq[.]	v2.05			230	DFP Encode BCD To DPD Quad X-form
111011 .... 11011 00010.	I	...X	DFP		diex[.]	v2.05			231	DFP Insert Biased Exponent X-form
111111 .... 11011 00010.	I	...X	DFP		diexq[.]	v2.05			231	DFP Insert Biased Exponent Quad X-form
011111 .... 01111 01001.	I	..XX			divd[.]	PPC		SR	87	Divide Doubleword XO-form
011111 .... 01101 01001.	I	..XX			divde[.]	v2.06		SR	88	Divide Doubleword Extended XO-form
011111 .... 11101 01001.	I	..XX			divdeo[.]	v2.06		SR	88	Divide Doubleword Extended & record OV XO-form
011111 .... 01100 01001.	I	..XX			divdeu[.]	v2.06		SR	88	Divide Doubleword Extended Unsigned XO-form
011111 .... 11100 01001.	I	..XX			divdeuo[.]	v2.06		SR	88	Divide Doubleword Extended Unsigned & record OV XO-form
011111 .... 11111 01001.	I	..XX			divdo[.]	PPC		SR	87	Divide Doubleword & record OV XO-form
011111 .... 01110 01001.	I	..XX			divdu[.]	PPC		SR	87	Divide Doubleword Unsigned XO-form
011111 .... 11110 01001.	I	..XX			divduo[.]	PPC		SR	87	Divide Doubleword Unsigned & record OV XO-form
011111 .... 01111 01011.	I	XXXX			divw[.]	PPC		SR	80	Divide Word XO-form
011111 .... 01101 01011.	I	XXXX			divwe[.]	v2.06		SR	81	Divide Word Extended XO-form
011111 .... 11101 01011.	I	..XX			divweo[.]	v2.06		SR	81	Divide Word Extended & record OV XO-form
011111 .... 01100 01011.	I	XXXX			divweu[.]	v2.06		SR	81	Divide Word Extended Unsigned XO-form
011111 .... 11100 01011.	I	..XX			divweuo[.]	v2.06		SR	81	Divide Word Extended Unsigned & record OV XO-form
011111 .... 11111 01011.	I	..XX			divwo[.]	PPC		SR	80	Divide Word & record OV XO-form
011111 .... 01110 01011.	I	XXXX			divwu[.]	PPC		SR	80	Divide Word Unsigned XO-form
011111 .... 11110 01011.	I	..XX			divwuo[.]	PPC		SR	80	Divide Word Unsigned & record OV XO-form
111011 .... 00001 00010.	I	...X	DFP		dmul[.]	v2.05			206	DFP Multiply X-form
111111 .... 00001 00010.	I	...X	DFP		dmulq[.]	v2.05			206	DFP Multiply Quad X-form
111011 ....000 00011.	I	...X	DFP		dqua[.]	v2.05			215	DFP Quantize Z23-form
111011 ....010 00011.	I	...X	DFP		dquai[.]	v2.05			214	DFP Quantize Immediate Z23-form
111111 ....010 00011.	I	...X	DFP		dquaiq[.]	v2.05			214	DFP Quantize Immediate Quad Z23-form
111111 ....000 00011.	I	...X	DFP		dquaq[.]	v2.05			215	DFP Quantize Quad Z23-form
111111 ..... ///// ..... 11000 00010.	I	...X	DFP		drdpq[.]	v2.05			226	DFP Round To DFP Long X-form
111011 ..... ////. ....111 00011.	I	...X	DFP		drintn[.]	v2.05			222	DFP Round To FP Integer Without Inexact Z23-form
111111 ..... ////. ....111 00011.	I	...X	DFP		drintnq[.]	v2.05			222	DFP Round To FP Integer Without Inexact Quad Z23-form
111011 ..... ////. ....011 00011.	I	...X	DFP		drintx[.]	v2.05			220	DFP Round To FP Integer With Inexact Z23-form
111111 ..... ////. ....011 00011.	I	...X	DFP		drintxq[.]	v2.05			220	DFP Round To FP Integer With Inexact Quad Z23-form

Table H.1: Power ISA Instruction Set Sorted by Mnemonic (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
111011 ....001 00011.	I	...X	DFP		drrnd[.]	v2.05			217	DFP Reround Z23-form
111111 ....001 00011.	I	...X	DFP		drrndq[.]	v2.05			217	DFP Reround Quad Z23-form
111011 ..... ///// ..... 11000 00010.	I	...X	DFP		drsp[.]	v2.05			226	DFP Round To DFP Short X-form
111011 .... .0010 00010.	I	...X	DFP		dscli[.]	v2.05			233	DFP Shift Significand Left Immediate Z22-form
111111 .... .0010 00010.	I	...X	DFP		dscliq[.]	v2.05			233	DFP Shift Significand Left Immediate Quad Z22-form
111011 .... .0011 00010.	I	...X	DFP		dscri[.]	v2.05			233	DFP Shift Significand Right Immediate Z22-form
111111 .... .0011 00010.	I	...X	DFP		dscriq[.]	v2.05			233	DFP Shift Significand Right Immediate Quad Z22-form
111011 .... 10000 00010.	I	...X	DFP		dsub[.]	v2.05			205	DFP Subtract X-form
111111 .... 10000 00010.	I	...X	DFP		dsubq[.]	v2.05			205	DFP Subtract Quad X-form
111011 ...// .... .0110 00010/	I	...X	DFP		dtstdc	v2.05			210	DFP Test Data Class Z22-form
111111 ...// .... .0110 00010/	I	...X	DFP		dtstdcq	v2.05			210	DFP Test Data Class Quad Z22-form
111011 ...// .... .0111 00010/	I	...X	DFP		dtstdg	v2.05			210	DFP Test Data Group Z22-form
111111 ...// .... .0111 00010/	I	...X	DFP		dtstdgq	v2.05			210	DFP Test Data Group Quad Z22-form
111011 ...// 00101 00010/	I	...X	DFP		dtstex	v2.05			211	DFP Test Exponent X-form
111111 ...// 00101 00010/	I	...X	DFP		dtstexq	v2.05			211	DFP Test Exponent Quad X-form
111011 ...// 10101 00010/	I	...X	DFP		dtstsf	v2.05			212	DFP Test Significance X-form
111011 ...// 10101 00011/	I	...X	DFP		dtstsfi	v3.0			213	DFP Test Significance Immediate X-form
111111 ...// 10101 00011/	I	...X	DFP		dtstsfiq	v3.0			213	DFP Test Significance Immediate Quad X-form
111111 ...// 10101 00010/	I	...X	DFP		dtstsfq	v2.05			212	DFP Test Significance Quad X-form
111011 ..... ///// ..... 01011 00010.	I	...X	DFP		dxex[.]	v2.05			231	DFP Extract Biased Exponent X-form
111111 ..... ///// ..... 01011 00010.	I	...X	DFP		dxexq[.]	v2.05			231	DFP Extract Biased Exponent Quad X-form
011111 ///// ///// ///// 11010 10110/	II	XXXX			eieio	PPC			1028	Enforce In-order Execution of I/O X-form
011111 .... 01000 11100.	I	XXXX			eqv[.]	P1		SR	100	Equivalent X-form
011111 .... ///// 11101 11010.	I	XXXX			extsb[.]	PPC		SR	101	Extend Sign Byte X-form
011111 .... ///// 11100 11010.	I	XXXX			extsh[.]	P1		SR	101	Extend Sign Halfword X-form
011111 .... ///// 11110 11010.	I	..XX			extsw[.]	PPC		SR	103	Extend Sign Word X-form
011111 .... 11011 1101..	I	..XX			extswsli[.]	v3.0			116	Extend Sign Word and Shift Left Immediate XS-form
111111 ..... ///// ..... 01000 01000.	I	.XXX			fabs[.]	P1			160	Floating Absolute Value X-form
111111 .... ///// 10101.	I	.XXX			fadd[.]	P1			162	Floating Add A-form
111011 .... ///// 10101.	I	.XXX			fadds[.]	PPC			162	Floating Add Single A-form
111111 ..... ///// ..... 11010 01110.	I	.XXX			fcfid[.]	PPC			175	Floating Convert with round Signed Doubleword to Double-Precision format X-form
111011 ..... ///// ..... 11010 01110.	I	.XXX			fcfids[.]	v2.06			176	Floating Convert with round Signed Doubleword to Single-Precision format X-form
111111 ..... ///// ..... 11110 01110.	I	.XXX			fcfidu[.]	v2.06			175	Floating Convert with round Unsigned Doubleword to Double-Precision format X-form
111011 ..... ///// ..... 11110 01110.	I	.XXX			fcfidus[.]	v2.06			176	Floating Convert with round Unsigned Doubleword to Single-Precision format X-form
111111 ...// 00001 00000/	I	.XXX			fcmpo	P1			179	Floating Compare Ordered X-form
111111 ...// 00000 00000/	I	.XXX			fcmpu	P1			179	Floating Compare Unordered X-form
111111 .... 00000 01000.	I	.XXX			fcpsgn[.]	v2.05			160	Floating Copy Sign X-form
111111 ..... ///// ..... 11001 01110.	I	.XXX			fctid[.]	PPC			171	Floating Convert with round Double-Precision To Signed Doubleword format X-form
111111 ..... ///// ..... 11101 01110.	I	.XXX			fctidu[.]	v2.06			172	Floating Convert with round Double-Precision To Unsigned Doubleword format X-form
111111 ..... ///// ..... 11101 01111.	I	.XXX			fctiduz[.]	v2.06			172	Floating Convert with truncate Double-Precision To Unsigned Doubleword format X-form
111111 ..... ///// ..... 11001 01111.	I	.XXX			fctidz[.]	PPC			171	Floating Convert with truncate Double-Precision To Signed Doubleword format X-form

Table H.1: Power ISA Instruction Set Sorted by Mnemonic (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
111111 ..... ///// ..... 00000 01110.	I	.XXX			fctiw[.]	P2			173	Floating Convert with round Double-Precision To Signed Word format X-form
111111 ..... ///// ..... 00100 01110.	I	.XXX			fctiwu[.]	v2.06			174	Floating Convert with round Double-Precision To Unsigned Word format X-form
111111 ..... ///// ..... 00100 01111.	I	.XXX			fctiwuz[.]	v2.06			174	Floating Convert with truncate Double-Precision To Unsigned Word format X-form
111111 ..... ///// ..... 00000 01111.	I	.XXX			fctiwz[.]	P2			173	Floating Convert with truncate Double-Precision To Signed Word format X-form
111111 .... ///// 10010.	I	.XXX			fdiv[.]	P1			163	Floating Divide A-form
111011 .... ///// 10010.	I	.XXX			fdivs[.]	PPC			163	Floating Divide Single A-form
111111 .... 11101.	I	.XXX			fmadd[.]	P1			168	Floating Multiply-Add A-form
111011 .... 11101.	I	.XXX			fmadds[.]	PPC			168	Floating Multiply-Add Single A-form
111111 ..... ///// ..... 00010 01000.	I	.XXX			fmr[.]	P1			160	Floating Move Register X-form
111111 .... 11110 00110/	I	.XXX			fmrgew	v2.07			161	Floating Merge Even Word X-form
111111 .... 11010 00110/	I	.XXX			fmrgow	v2.07			161	Floating Merge Odd Word X-form
111111 .... 11100.	I	.XXX			fmsub[.]	P1			168	Floating Multiply-Subtract A-form
111011 .... 11100.	I	.XXX			fmsubs[.]	PPC			168	Floating Multiply-Subtract Single A-form
111111 .... ///// ..... 11001.	I	.XXX			fmul[.]	P1			163	Floating Multiply A-form
111011 .... ///// ..... 11001.	I	.XXX			fmuls[.]	PPC			163	Floating Multiply Single A-form
111111 ..... ///// ..... 00100 01000.	I	.XXX			fnabs[.]	P1			160	Floating Negative Absolute Value X-form
111111 ..... ///// ..... 00001 01000.	I	.XXX			fneg[.]	P1			160	Floating Negate X-form
111111 .... 11111.	I	.XXX			fnmadd[.]	P1			169	Floating Negative Multiply-Add A-form
111011 .... 11111.	I	.XXX			fnmadds[.]	PPC			169	Floating Negative Multiply-Add Single A-form
111111 .... 11110.	I	.XXX			fnmsub[.]	P1			169	Floating Negative Multiply-Subtract A-form
111011 .... 11110.	I	.XXX			fnmsubs[.]	PPC			169	Floating Negative Multiply-Subtract Single A-form
111111 ..... ///// ..... ///// 11000.	I	.XXX			fre[.]	v2.02			165	Floating Reciprocal Estimate A-form
111011 ..... ///// ..... ///// 11000.	I	.XXX			fres[.]	PPC			165	Floating Reciprocal Estimate Single A-form
111111 ..... ///// ..... 01111 01000.	I	.XXX			frim[.]	v2.02			178	Floating Round to Integer Minus X-form
111111 ..... ///// ..... 01100 01000.	I	.XXX			frin[.]	v2.02			177	Floating Round to Integer Nearest X-form
111111 ..... ///// ..... 01110 01000.	I	.XXX			frip[.]	v2.02			178	Floating Round to Integer Plus X-form
111111 ..... ///// ..... 01101 01000.	I	.XXX			friz[.]	v2.02			177	Floating Round to Integer Toward Zero X-form
111111 ..... ///// ..... 00000 01100.	I	.XXX			frsp[.]	P1			170	Floating Round to Single-Precision X-form
111111 ..... ///// ..... ///// 11010.	I	.XXX			frsqrte[.]	PPC			166	Floating Reciprocal Square Root Estimate A-form
111011 ..... ///// ..... ///// 11010.	I	.XXX			frsqrtes[.]	v2.02			166	Floating Reciprocal Square Root Estimate Single A-form
111111 .... 10111.	I	.XXX			fsel[.]	PPC			180	Floating Select A-form
111111 ..... ///// ..... ///// 10110.	I	.XXX			fsqrt[.]	P2			164	Floating Square Root A-form
111011 ..... ///// ..... ///// 10110.	I	.XXX			fsqrts[.]	PPC			164	Floating Square Root Single A-form
111111 .... ///// 10100.	I	.XXX			fsub[.]	P1			162	Floating Subtract A-form
111011 .... ///// 10100.	I	.XXX			fsubs[.]	PPC			162	Floating Subtract Single A-form
111111 .../// 00100 00000/	I	.XXX			ftdiv	v2.06			167	Floating Test for software Divide X-form
111111 .../// ///// 00101 00000/	I	.XXX			ftsqrt	v2.06			167	Floating Test for software Square Root X-form
011111 .... 10111 10010.	I	....			hashchk	v3.1B			120	Hash Check X-form
011111 .... 10101 10010.	III	....			hashchkp	v3.1B	P		1099	Hash Check Privileged X-form
011111 .... 10110 10010.	I	....			hashst	v3.1B			120	Hash Store X-form
011111 .... 10100 10010.	III	....			hashstp	v3.1B	P		1099	Hash Store Privileged X-form
010011 ///// ///// ///// 01000 10010/	III	..XX			hrfid	v2.02	HV		1086	Hypervisor Return From Interrupt Doubleword XL-form
011111 ///// 11110 10110/	II	..XX			icbi	PPC			991	Instruction Cache Block Invalidate X-form
011111 /.... .... 00000 10110/	II	..XX			icbt	v2.07			991	Instruction Cache Block Touch X-form

Table H.1: Power ISA Instruction Set Sorted by Mnemonic (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
011111 .... 01111/	I	XXXX			isel	v2.03			96	Integer Select A-form
010011 ///// ///// ///// 00100 10110/	II	XXXX			isync	P1			1014	Instruction Synchronize XL-form
011111 .... 00001 10100.	II	XXXX			lbarx	v2.06			1016	Load Byte And Reserve Indexed X-form
100010	I	XXXX			lbz	P1			48	Load Byte and Zero D-form
011111 .... 11010 10101/	III	..XX			lbzcix	v2.05	HV		1097	Load Byte and Zero Caching Inhibited Indexed X-form
100011	I	XXXX			lbzu	P1			49	Load Byte and Zero with Update D-form
011111 .... 00011 10111/	I	XXXX			lbzux	P1			49	Load Byte and Zero with Update Indexed X-form
011111 .... 00010 10111/	I	XXXX			lbzx	P1			48	Load Byte and Zero Indexed X-form
111010 00	I	..XX			ld	PPC			55	Load Doubleword DS-form
011111 .... 00010 10100.	II	..XX			ldarx	PPC			1021	Load Doubleword And Reserve Indexed X-form
011111 .... 10011 00110/	II	...X	AMO		ldat	v3.0			1011	Load Doubleword Atomic X-form
011111 .... 10000 10100/	I	..XX			ldbrx	v2.06			67	Load Doubleword Byte-Reverse Indexed X-form
011111 .... 11011 10101/	III	..XX			ldcix	v2.05	HV		1097	Load Doubleword Caching Inhibited Indexed X-form
111010 01	I	..XX			ldu	PPC			55	Load Doubleword with Update DS-form
011111 .... 00001 10101/	I	..XX			ldux	PPC			55	Load Doubleword with Update Indexed X-form
011111 .... 00000 10101/	I	..XX			ldx	PPC			55	Load Doubleword Indexed X-form
110010	I	.XXX			lfd	P1			152	Load Floating-Point Double D-form
111001 00	I	...X			lfdp	v2.05			158	Load Floating-Point Double Pair DS-form
011111 .... 11000 10111/	I	...X			lfdpx	v2.05			158	Load Floating-Point Double Pair Indexed X-form
110011	I	.XXX			lfdu	P1			153	Load Floating-Point Double with Update D-form
011111 .... 10011 10111/	I	.XXX			lfdux	P1			153	Load Floating-Point Double with Update Indexed X-form
011111 .... 10010 10111/	I	.XXX			lfdx	P1			152	Load Floating-Point Double Indexed X-form
011111 .... 11010 10111/	I	.XXX			lfiwax	v2.05			153	Load Floating-Point as Integer Word Algebraic Indexed X-form
011111 .... 11011 10111/	I	.XXX			lfiwzx	v2.06			153	Load Floating-Point as Integer Word & Zero Indexed X-form
110000	I	.XXX			lfs	P1			150	Load Floating-Point Single D-form
110001	I	.XXX			lfsu	P1			151	Load Floating-Point Single with Update D-form
011111 .... 10001 10111/	I	.XXX			lfsux	P1			151	Load Floating-Point Single with Update Indexed X-form
011111 .... 10000 10111/	I	.XXX			lfsx	P1			150	Load Floating-Point Single Indexed X-form
101010	I	XXXX			lha	P1			51	Load Halfword Algebraic D-form
011111 .... 00011 10100.	II	XXXX			lharx	v2.06			1016	Load Halfword And Reserve Indexed X-form
101011	I	XXXX			lhau	P1			52	Load Halfword Algebraic with Update D-form
011111 .... 01011 10111/	I	XXXX			lhaux	P1			52	Load Halfword Algebraic with Update Indexed X-form
011111 .... 01010 10111/	I	XXXX			lhax	P1			52	Load Halfword Algebraic Indexed X-form
011111 .... 11000 10110/	I	XXXX			lhbrx	P1			66	Load Halfword Byte-Reverse Indexed X-form
101000	I	XXXX			lhz	P1			50	Load Halfword and Zero D-form
011111 .... 11001 10101/	III	..XX			lhzcix	v2.05	HV		1097	Load Halfword and Zero Caching Inhibited Indexed X-form
101001	I	XXXX			lhzu	P1			51	Load Halfword and Zero with Update D-form
011111 .... 01001 10111/	I	XXXX			lhzux	P1			51	Load Halfword and Zero with Update Indexed X-form
011111 .... 01000 10111/	I	XXXX			lhzx	P1			50	Load Halfword and Zero Indexed X-form
101110	I	...X			lmw	P1			68	Load Multiple Word D-form
111000	I	..XX			lq	v2.03			63	Load Quadword DQ-form
011111 .... 01000 10100.	I	..XX			lqarx	v2.07			1023	Load Quadword And Reserve Indexed X-form
011111 .... 10010 10101/	I	...X			lswi	P1			70	Load String Word Immediate X-form
011111 .... 10000 10101/	I	...X			lswx	P1			70	Load String Word Indexed X-form
011111 .... 00000 00111/	I	..XX			lvebx	v2.03			260	Load Vector Element Byte Indexed X-form

Table H.1: Power ISA Instruction Set Sorted by Mnemonic (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
011111 .... 00001 00111/	I	..XX			lvehx	v2.03			260	Load Vector Element Halfword Indexed X-form
011111 .... 00010 00111/	I	..XX			lvewx	v2.03			261	Load Vector Element Word Indexed X-form
011111 .... 00000 00110/	I	..XX			lvsl	v2.03			267	Load Vector for Shift Left Indexed X-form
011111 .... 00001 00110/	I	..XX			lvsr	v2.03			267	Load Vector for Shift Right Indexed X-form
011111 .... 00011 00111/	I	..XX			lvx	v2.03			262	Load Vector Indexed X-form
011111 .... 01011 00111/	I	..XX			lvxl	v2.03			262	Load Vector Indexed Last X-form
111010 10	I	..XX			lwa	PPC			54	Load Word Algebraic DS-form
011111 .... 00000 10100.	II	XXXX			lwarx	PPC			1017	Load Word And Reserve Indexed X-form
011111 .... 10010 00110/	II	...X	AMO		lwat	v3.0			1011	Load Word Atomic X-form
011111 .... 01011 10101/	I	..XX			lwaux	PPC			54	Load Word Algebraic with Update Indexed X-form
011111 .... 01010 10101/	I	..XX			lwax	PPC			54	Load Word Algebraic Indexed X-form
011111 .... 10000 10110/	I	XXXX			lwbrx	P1			66	Load Word Byte-Reverse Indexed X-form
100000	I	XXXX			lwz	P1			53	Load Word and Zero D-form
011111 .... 11000 10101/	III	..XX			lwzcix	v2.05	HV		1097	Load Word and Zero Caching Inhibited Indexed X-form
100001	I	XXXX			lwzu	P1			53	Load Word and Zero with Update D-form
011111 .... 00001 10111/	I	XXXX			lwzux	P1			53	Load Word and Zero with Update Indexed X-form
011111 .... 00000 10111/	I	XXXX			lwzx	P1			53	Load Word and Zero Indexed X-form
111001 10	I	..XX			lxsd	v3.0			562	Load VSX Scalar Doubleword DS-form
011111 .... 10010 01100.	I	..XX			lxsdx	v2.06			563	Load VSX Scalar Doubleword Indexed X-form
011111 .... 11000 01101.	I	..XX			lxsibzx	v3.0			564	Load VSX Scalar as Integer Byte & Zero Indexed X-form
011111 .... 11001 01101.	I	..XX			lxsihzx	v3.0			564	Load VSX Scalar as Integer Halfword & Zero Indexed X-form
011111 .... 00010 01100.	I	..XX			lxsiwax	v2.07			565	Load VSX Scalar as Integer Word Algebraic Indexed X-form
011111 .... 00000 01100.	I	..XX			lxsiwzx	v2.07			566	Load VSX Scalar as Integer Word & Zero Indexed X-form
111001 11	I	..XX			lxssp	v3.0			567	Load VSX Scalar Single-Precision DS-form
011111 .... 10000 01100.	I	..XX			lxsspx	v2.07			568	Load VSX Scalar Single-Precision Indexed X-form
111101 001	I	..XX			lxv	v3.0			575	Load VSX Vector DQ-form
011111 .... 11011 01100.	I	..XX			lxvb16x	v3.0			576	Load VSX Vector Byte*16 Indexed X-form
011111 .... 11010 01100.	I	..XX			lxvd2x	v2.06			577	Load VSX Vector Doubleword*2 Indexed X-form
011111 .... 01010 01100.	I	..XX			lxvdsx	v2.06			582	Load VSX Vector Doubleword & Splat Indexed X-form
011111 .... 11001 01100.	I	..XX			lxvh8x	v3.0			578	Load VSX Vector Halfword*8 Indexed X-form
111100 ..... 11111 ..... 01011 01000.	I	..XX			lxvkq	v3.1			935	Load VSX Vector Special Value Quadword X-form
011111 .... 01000 01101.	I	..XX			lxvl	v3.0			588	Load VSX Vector with Length X-form
011111 .... 01001 01101.	I	..XX			lxvll	v3.0			590	Load VSX Vector with Length Left-justified X-form
000110 0000	I	..XX			lxvp	v3.1			603	Load VSX Vector Paired DQ-form
011111 .... 01010 01101/	I	..XX			lxvpx	v3.1			604	Load VSX Vector Paired Indexed X-form
011111 .... 00000 01101.	I	..XX			lxvrbx	v3.1			584	Load VSX Vector Rightmost Byte Indexed X-form
011111 .... 00011 01101.	I	..XX			lxvrdx	v3.1			585	Load VSX Vector Rightmost Doubleword Indexed X-form
011111 .... 00001 01101.	I	..XX			lxvrhx	v3.1			586	Load VSX Vector Rightmost Halfword Indexed X-form
011111 .... 00010 01101.	I	..XX			lxvrwx	v3.1			587	Load VSX Vector Rightmost Word Indexed X-form
011111 .... 11000 01100.	I	..XX			lxvw4x	v2.06			579	Load VSX Vector Word*4 Indexed X-form
011111 .... 01011 01100.	I	..XX			lxvwsx	v3.0			583	Load VSX Vector Word & Splat Indexed X-form
011111 .... 0100/ 01100.	I	..XX			lxvx	v3.0			580	Load VSX Vector Indexed X-form
000100 110000	I	..XX			maddhd	v3.0			86	Multiply-Add High Doubleword VA-form
000100 110001	I	..XX			maddhdu	v3.0			86	Multiply-Add High Doubleword Unsigned VA-form
000100 110011	I	..XX			maddld	v3.0			86	Multiply-Add Low Doubleword VA-form
010011 .../// .../// ///// 00000 00000/	I	XXXX			mcrf	P1			45	Move Condition Register Field XL-form

Table H.1: Power ISA Instruction Set Sorted by Mnemonic (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
111111 ...// ...// ///// 00010 00000/	I	.XXX			mcrfs	P1			184	Move to Condition Register from FPSCR X-form
011111 ...// ///// ///// 10010 00000/	I	XXXX			mcrxrx	v3.0			129	Move to CR from XER Extended X-form
011111 .... 01001 01110/	I	...X	BHRB		mfbhrbe	v2.07			1041	Move From Branch History Rolling Buffer Entry XFX-form
011111 ..... 0//// ///// 00000 10011/	I	XXXX			mfcr	P1			130	Move From Condition Register XFX-form
111111 ..... 00000 ///// 10010 00111.	I	.XXX			mffs[.]	P1			182	Move From FPSCR X-form
111111 ..... 10100 ..... 10010 00111/	I	.XXX			mffscdrn	v3.0B			183	Move From FPSCR Control & Set DRN X-form
111111 ..... 10101 //... 10010 00111/	I	.XXX			mffscdrni	v3.0B			183	Move From FPSCR Control & Set DRN Immediate X-form
111111 ..... 00001 ///// 10010 00111/	I	.XXX			mffsce	v3.0B			182	Move From FPSCR & Clear Enables X-form
111111 ..... 10110 ..... 10010 00111/	I	.XXX			mffscrn	v3.0B			183	Move From FPSCR Control & Set RN X-form
111111 ..... 10111 ///.. 10010 00111/	I	.XXX			mffscrni	v3.0B			183	Move From FPSCR Control & Set RN Immediate X-form
111111 ..... 11000 ///// 10010 00111/	I	.XXX			mffsl	v3.0B			184	Move From FPSCR Lightweight X-form
011111 ..... ///// ///// 00010 10011/	III	XXXX			mfmsr	P1	P		1110	Move From Machine State Register X-form
011111 ..... 1.... ..../ 00000 10011/	I	XXXX			mfocrf	v2.01			130	Move From One Condition Register Field XFX-form
011111 .... 01010 10011/	I III	XXXX			mfspr	P1	O		128	Move From Special Purpose Register XFX-form
011111 .... 01011 10011/	II	XXXX			mftb	PPC			1032	Move From Time Base XFX-form
000100 ..... ///// ///// 11000 000100	I	..XX			mfvscr	v2.03			483	Move From Vector Status and Control Register VX-form
011111 .... ///// 00001 10011.	I	..XX			mfvsrd	v2.07			122	Move From VSR Doubleword X-form
011111 .... ///// 01001 10011.	I	..XX			mfvsrld	v3.0			122	Move From VSR Lower Doubleword X-form
011111 .... ///// 00011 10011.	I	..XX			mfvsrwz	v2.07			123	Move From VSR Word and Zero X-form
011111 .... 11000 01001/	I	..XX			modsd	v3.0			89	Modulo Signed Doubleword X-form
011111 .... 11000 01011/	I	XXXX			modsw	v3.0			83	Modulo Signed Word X-form
011111 .... 01000 01001/	I	..XX			modud	v3.0			89	Modulo Unsigned Doubleword X-form
011111 .... 01000 01011/	I	XXXX			moduw	v3.0			83	Modulo Unsigned Word X-form
011111 ///// ///// 00111 01110/	III	..XX			msgclr	v2.07	HV		1275	Message Clear X-form
011111 ///// ///// 00101 01110/	III	..XX			msgclrp	v2.07	P		1276	Message Clear Privileged X-form
011111 ///// ///// 00011 01110/	III	....			msgclru	v3.0C	UV		1274	Message Clear Ultravisor X-form
011111 ///// ///// 00110 01110/	III	..XX			msgsnd	v2.07	HV		1275	Message Send X-form
011111 ///// ///// 00100 01110/	III	..XX			msgsndp	v2.07	P		1276	Message Send Privileged X-form
011111 ///// ///// 00010 01110/	III	....			msgsndu	v3.0C	UV		1274	Message Send Ultravisor X-form
011111 ///// ///// ///// 11011 10110/	III	..XX			msgsync	v3.0	HV		1277	Message Synchronize X-form
011111 ..... 0.... ..../ 00100 10000/	I	XXXX			mtcrf	P1			129	Move To Condition Register Fields XFX-form
111111 ..... ///// ///// 00010 00110.	I	.XXX			mtfsb0[.]	P1			185	Move To FPSCR Bit 0 X-form
111111 ..... ///// ///// 00001 00110.	I	.XXX			mtfsb1[.]	P1			185	Move To FPSCR Bit 1 X-form
111111 .... 10110 00111.	I	.XXX			mtfsf[.]	P1			185	Move To FPSCR Fields XFL-form
111111 ...// ////. ..../ 00100 00110.	I	.XXX			mtfsfi[.]	P1			184	Move To FPSCR Field Immediate X-form
011111 ..... ////. ///// 00100 10010/	III	XXXX			mtmsr	P1	P		1108	Move To Machine State Register X-form
011111 ..... ////. ///// 00101 10010/	III	..XX			mtmsrd	PPC	P		1109	Move To Machine State Register Doubleword X-form
011111 ..... 1.... ..../ 00100 10000/	I	XXXX			mtocrf	v2.01			129	Move To One Condition Register Field XFX-form
011111 .... 01110 10011/	I III	XXXX			mtspr	P1	O		126	Move To Special Purpose Register XFX-form
000100 ///// ///// 11001 000100	I	..XX			mtvscr	v2.03			483	Move To Vector Status and Control Register VX-form
000100 ..... 10000 ..... 11001 000010	I	..XX			mtvsrbm	v3.1			451	Move to VSR Byte Mask VX-form
000100 .... 01010.	I	..XX			mtvsrbmi	v3.1			453	Move To VSR Byte Mask Immediate DX-form
011111 .... ///// 00101 10011.	I	..XX			mtvsrd	v2.07			123	Move To VSR Doubleword X-form
011111 .... 01101 10011.	I	..XX			mtvsrdd	v3.0			125	Move To VSR Double Doubleword X-form
000100 ..... 10011 ..... 11001 000010	I	..XX			mtvsrdm	v3.1			452	Move to VSR Doubleword Mask VX-form
000100 ..... 10001 ..... 11001 000010	I	..XX			mtvsrhm	v3.1			451	Move to VSR Halfword Mask VX-form
000100 ..... 10100 ..... 11001 000010	I	..XX			mtvsrqm	v3.1			453	Move to VSR Quadword Mask VX-form

Table H.1: Power ISA Instruction Set Sorted by Mnemonic (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
011111 .... ///// 00110 10011.	I	..XX			mtvsrwa	v2.07			124	Move To VSR Word Algebraic X-form
000100 ..... 10010 ..... 11001 000010	I	..XX			mtvsrwm	v3.1			452	Move to VSR Word Mask VX-form
011111 .... ///// 01100 10011.	I	..XX			mtvsrws	v3.0			125	Move To VSR Word & Splat X-form
011111 .... ///// 00111 10011.	I	..XX			mtvsrwz	v2.07			124	Move To VSR Word and Zero X-form
011111 .... /0010 01001.	I	..XX			mulhd[.]	PPC		SR	85	Multiply High Doubleword XO-form
011111 .... /0000 01001.	I	..XX			mulhdu[.]	PPC		SR	85	Multiply High Doubleword Unsigned XO-form
011111 .... /0010 01011.	I	XXXX			mulhw[.]	PPC		SR	79	Multiply High Word XO-form
011111 .... /0000 01011.	I	XXXX			mulhwu[.]	PPC		SR	79	Multiply High Word Unsigned XO-form
011111 .... 00111 01001.	I	..XX			mulld[.]	PPC		SR	85	Multiply Low Doubleword XO-form
011111 .... 10111 01001.	I	..XX			mulldo[.]	PPC		SR	85	Multiply Low Doubleword & record OV XO-form
000111	I	XXXX			mulli	P1			79	Multiply Low Immediate D-form
011111 .... 00111 01011.	I	XXXX			mullw[.]	P1		SR	79	Multiply Low Word XO-form
011111 .... 10111 01011.	I	..XX			mullwo[.]	P1		SR	79	Multiply Low Word & record OV XO-form
011111 .... 01110 11100.	I	XXXX			nand[.]	P1		SR	99	NAND X-form
011111 .... ///// 00011 01000.	I	XXXX			neg[.]	P1		SR	78	Negate XO-form
011111 .... ///// 10011 01000.	I	..XX			nego[.]	P1		SR	78	Negate & record OV XO-form
011111 .... 00011 11100.	I	XXXX			nor[.]	P1		SR	100	NOR X-form
011111 .... 01101 11100.	I	XXXX			or[.]	P1		SR	99	OR X-form
011111 .... 01100 11100.	I	XXXX			orc[.]	P1		SR	100	OR with Complement X-form
011000	I	XXXX			ori	P1			98	OR Immediate D-form
011001	I	XXXX			oris	P1			98	OR Immediate Shifted D-form
000001 100// .//.. .... 001110	I	..XX			paddi	v3.1			73	Prefixed Add Immediate MLS:D-form
011111 ////. .... 11100 00110.	II	....			paste.	v3.0			1008	Paste X-form
011111 .... 00100 11100/	I	..XX			pdepd	v3.1			106	Parallel Bits Deposit Doubleword X-form
011111 .... 00101 11100/	I	..XX			pextd	v3.1			106	Parallel Bits Extract Doubleword X-form
000001 100// .//.. .... 100010	I	..XX			plbz	v3.1			48	Prefixed Load Byte and Zero MLS:D-form
000001 000// .//.. .... 111001	I	..XX			pld	v3.1			55	Prefixed Load Doubleword 8LS:D-form
000001 100// .//.. .... 110010	I	..XX			plfd	v3.1			152	Prefixed Load Floating-Point Double MLS:D-form
000001 100// .//.. .... 110000	I	..XX			plfs	v3.1			150	Prefixed Load Floating-Point Single MLS:D-form
000001 100// .//.. .... 101010	I	..XX			plha	v3.1			51	Prefixed Load Halfword Algebraic MLS:D-form
000001 100// .//.. .... 101000	I	..XX			plhz	v3.1			50	Prefixed Load Halfword and Zero MLS:D-form
000001 000// .//.. .... 111000	I	..XX			plq	v3.1			63	Prefixed Load Quadword 8LS:D-form
000001 000// .//.. .... 101001	I	..XX			plwa	v3.1			54	Prefixed Load Word Algebraic 8LS:D-form
000001 100// .//.. .... 100000	I	..XX			plwz	v3.1			53	Prefixed Load Word and Zero MLS:D-form
000001 000// .//.. .... 101010	I	..XX			plxsd	v3.1			562	Prefixed Load VSX Scalar Doubleword 8LS:D-form
000001 000// .//.. .... 101011	I	..XX			plxssp	v3.1			567	Prefixed Load VSX Scalar Single-Precision 8LS:D-form

Table H.1: Power ISA Instruction Set Sorted by Mnemonic (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
000001 000// .//.. .... 11001.	I	..XX			plxv	v3.1			575	Prefixed Load VSX Vector 8LS:D-form
000001 000// .//.. .... 111010	I	..XX			plxvp	v3.1			603	Prefixed Load VSX Vector Paired 8LS:D-form
000001 11100 1//// ..//// ///.. ..... 111011 ...// .... 00110 011../	I	....	MMA	MMA	pmxvbf16ger2	v3.1			889	Prefixed Masked VSX Vector bfloat16 GER (rank-2 update) MMIRR:XX3-form
000001 11100 1//// ..//// ///.. ..... 111011 ...// .... 11110 010../	I	....	MMA	MMA	pmxvbf16ger2nn	v3.1			890	Prefixed Masked VSX Vector bfloat16 GER (rank-2 update) Negative multiply, Negative accumulate MMIRR:XX3-form
000001 11100 1//// ..//// ///.. ..... 111011 ...// .... 01110 010../	I	....	MMA	MMA	pmxvbf16ger2np	v3.1			890	Prefixed Masked VSX Vector bfloat16 GER (rank-2 update) Negative multiply, Positive accumulate MMIRR:XX3-form
000001 11100 1//// ..//// ///.. ..... 111011 ...// .... 10110 010../	I	....	MMA	MMA	pmxvbf16ger2pn	v3.1			889	Prefixed Masked VSX Vector bfloat16 GER (rank-2 update) Positive multiply, Negative accumulate MMIRR:XX3-form
000001 11100 1//// ..//// ///.. ..... 111011 ...// .... 00110 010../	I	....	MMA	MMA	pmxvbf16ger2pp	v3.1			889	Prefixed Masked VSX Vector bfloat16 GER (rank-2 update) Positive multiply, Positive accumulate MMIRR:XX3-form
000001 11100 1//// ..//// ///.. ..... 111011 ...// .... 00010 011../	I	....	MMA	MMA	pmxvf16ger2	v3.1			894	Prefixed Masked VSX Vector 16-bit Floating-Point GER (rank-2 update) MMIRR:XX3-form
000001 11100 1//// ..//// ///.. ..... 111011 ...// .... 11010 010../	I	....	MMA	MMA	pmxvf16ger2nn	v3.1			895	Prefixed Masked VSX Vector 16-bit Floating-Point GER (rank-2 update) Negative multiply, Negative accumulate MMIRR:XX3-form
000001 11100 1//// ..//// ///.. ..... 111011 ...// .... 01010 010../	I	....	MMA	MMA	pmxvf16ger2np	v3.1			895	Prefixed Masked VSX Vector 16-bit Floating-Point GER (rank-2 update) Negative multiply, Positive accumulate MMIRR:XX3-form
000001 11100 1//// ..//// ///.. ..... 111011 ...// .... 10010 010../	I	....	MMA	MMA	pmxvf16ger2pn	v3.1			894	Prefixed Masked VSX Vector 16-bit Floating-Point GER (rank-2 update) Positive multiply, Negative accumulate MMIRR:XX3-form
000001 11100 1//// ..//// ///.. ..... 111011 ...// .... 00010 010../	I	....	MMA	MMA	pmxvf16ger2pp	v3.1			894	Prefixed Masked VSX Vector 16-bit Floating-Point GER (rank-2 update) Positive multiply, Positive accumulate MMIRR:XX3-form
000001 11100 1//// ///// ///.. ..... 111011 ...// .... 00011 011../	I	....	MMA	MMA	pmxvf32ger	v3.1			899	Prefixed Masked VSX Vector 32-bit Floating-Point GER (rank-1 update) MMIRR:XX3-form
000001 11100 1//// ///// ///.. ..... 111011 ...// .... 11011 010../	I	....	MMA	MMA	pmxvf32gernn	v3.1			900	Prefixed Masked VSX Vector 32-bit Floating-Point GER (rank-1 update) Negative multiply, Negative accumulate MMIRR:XX3-form
000001 11100 1//// ///// ///.. ..... 111011 ...// .... 01011 010../	I	....	MMA	MMA	pmxvf32gernp	v3.1			900	Prefixed Masked VSX Vector 32-bit Floating-Point GER (rank-1 update) Negative multiply, Positive accumulate MMIRR:XX3-form
000001 11100 1//// ///// ///.. ..... 111011 ...// .... 10011 010../	I	....	MMA	MMA	pmxvf32gerpn	v3.1			899	Prefixed Masked VSX Vector 32-bit Floating-Point GER (rank-1 update) Positive multiply, Negative accumulate MMIRR:XX3-form
000001 11100 1//// ///// ///.. ..... 111011 ...// .... 00011 010../	I	....	MMA	MMA	pmxvf32gerpp	v3.1			899	Prefixed Masked VSX Vector 32-bit Floating-Point GER (rank-1 update) Positive multiply, Positive accumulate MMIRR:XX3-form
000001 11100 1//// ///// ///.. ..... 111011 ...// .... 00111 011../	I	....	MMA	MMA	pmxvf64ger	v3.1			903	Prefixed Masked VSX Vector 64-bit Floating-Point GER (rank-1 update) MMIRR:XX3-form

Table H.1: Power ISA Instruction Set Sorted by Mnemonic (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
000001 11100 1//// ///// ///.. ..... 111011 ...// .... 11111 010../	I	....	MMA	MMA	pmxvf64gernn	v3.1			904	Prefixed Masked VSX Vector 64-bit Floating-Point GER (rank-1 update) Negative multiply, Negative accumulate MMIRR:XX3-form
000001 11100 1//// ///// ///.. ..... 111011 ...// .... 01111 010../	I	....	MMA	MMA	pmxvf64gernp	v3.1			904	Prefixed Masked VSX Vector 64-bit Floating-Point GER (rank-1 update) Negative multiply, Positive accumulate MMIRR:XX3-form
000001 11100 1//// ///// ///.. ..... 111011 ...// .... 10111 010../	I	....	MMA	MMA	pmxvf64gerpn	v3.1			903	Prefixed Masked VSX Vector 64-bit Floating-Point GER (rank-1 update) Positive multiply, Negative accumulate MMIRR:XX3-form
000001 11100 1//// ///// ///.. ..... 111011 ...// .... 00111 010../	I	....	MMA	MMA	pmxvf64gerpp	v3.1			903	Prefixed Masked VSX Vector 64-bit Floating-Point GER (rank-1 update) Positive multiply, Positive accumulate MMIRR:XX3-form
000001 11100 1//// ..//// ///.. ..... 111011 ...// .... 01001 011../	I	....	MMA	MMA	pmxvi16ger2	v3.1			878	Prefixed Masked VSX Vector 16-bit Signed Integer GER (rank-2 update) MMIRR:XX3-form
000001 11100 1//// ..//// ///.. ..... 111011 ...// .... 01101 011../	I	....	MMA	MMA	pmxvi16ger2pp	v3.1			878	Prefixed Masked VSX Vector 16-bit Signed Integer GER (rank-2 update) Positive multiply, Positive accumulate MMIRR:XX3-form
000001 11100 1//// ..//// ///.. ..... 111011 ...// .... 00101 011../	I	....	MMA	MMA	pmxvi16ger2s	v3.1			880	Prefixed Masked VSX Vector 16-bit Signed Integer GER (rank-2 update) with Saturation MMIRR:XX3-form
000001 11100 1//// ..//// ///.. ..... 111011 ...// .... 00101 010../	I	....	MMA	MMA	pmxvi16ger2spp	v3.1			880	Prefixed Masked VSX Vector 16-bit Signed Integer GER (rank-2 update) with Saturation Positive multiply, Positive accumulate MMIRR:XX3-form
000001 11100 1//// .... 111011 ...// .... 00100 011../	I	....	MMA	MMA	pmxvi4ger8	v3.1			882	Prefixed Masked VSX Vector 4-bit Signed Integer GER (rank-8 update) MMIRR:XX3-form
000001 11100 1//// .... 111011 ...// .... 00100 010../	I	....	MMA	MMA	pmxvi4ger8pp	v3.1			882	Prefixed Masked VSX Vector 4-bit Signed Integer GER (rank-8 update) Positive multiply, Positive accumulate MMIRR:XX3-form
000001 11100 1//// ....// ///.. ..... 111011 ...// .... 00000 011../	I	....	MMA	MMA	pmxvi8ger4	v3.1			885	Prefixed Masked VSX Vector 8-bit Signed/Unsigned Integer GER (rank-4 update) MMIRR:XX3-form
000001 11100 1//// ....// ///.. ..... 111011 ...// .... 00000 010../	I	....	MMA	MMA	pmxvi8ger4pp	v3.1			885	Prefixed Masked VSX Vector 8-bit Signed/Unsigned Integer GER (rank-4 update) Positive multiply, Positive accumulate MMIRR:XX3-form
000001 11100 1//// ....// ///.. ..... 111011 ...// .... 01100 011../	I	....	MMA	MMA	pmxvi8ger4spp	v3.1			887	Prefixed Masked VSX Vector 8-bit Signed/Unsigned Integer GER (rank-4 update) with Saturation Positive multiply, Positive accumulate MMIRR:XX3-form
000001 11000 0//00 00000 00000 000000 ?????? ????? ????? ????? ????? ??????	I	..XX			pnop	v3.1			132	Prefixed Nop MRR:*-form
011111 .... ///// 00011 11010/	I	XXXX			popcntb	v2.02			102	Population Count Bytes X-form
011111 .... ///// 01111 11010/	I	..XX			popcntd	v2.06			103	Population Count Doubleword X-form
011111 .... ///// 01011 11010/	I	XXXX			popcntw	v2.06			102	Population Count Words X-form
011111 .... ///// 00101 11010/	I	..XX			prtyd	v2.05			103	Parity Doubleword X-form
011111 .... ///// 00100 11010/	I	XXXX			prtyw	v2.05			102	Parity Word X-form
000001 100// .//.. .... 100110	I	..XX			pstb	v3.1			57	Prefixed Store Byte MLS:D-form
000001 000// .//.. .... 111101	I	..XX			pstd	v3.1			60	Prefixed Store Doubleword 8LS:D-form
000001 100// .//.. .... 110110	I	..XX			pstfd	v3.1			156	Prefixed Store Floating-Point Double MLS:D-form

Table H.1: Power ISA Instruction Set Sorted by Mnemonic (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
000001 100// .//.. .... 110100	I	..XX			pstfs	v3.1			154	Prefixed Store Floating-Point Single MLS:D-form
000001 100// .//.. .... 101100	I	..XX			psth	v3.1			58	Prefixed Store Halfword MLS:D-form
000001 000// .//.. .... 111100	I	..XX			pstq	v3.1			64	Prefixed Store Quadword 8LS:D-form
000001 100// .//.. .... 100100	I	..XX			pstw	v3.1			59	Prefixed Store Word MLS:D-form
000001 000// .//.. .... 101110	I	..XX			pstxsd	v3.1			569	Prefixed Store VSX Scalar Doubleword 8LS:D-form
000001 000// .//.. .... 101111	I	..XX			pstxssp	v3.1			573	Prefixed Store VSX Scalar Single-Precision 8LS:D-form
000001 000// .//.. .... 11011.	I	..XX			pstxv	v3.1			591	Prefixed Store VSX Vector 8LS:D-form
000001 000// .//.. .... 111110	I	..XX			pstxvp	v3.1			605	Prefixed Store VSX Vector Paired 8LS:D-form
010011 ///// ///// ////. 00100 10010/	I	...X	EBB		rfebb	v2.07			1037	Return from Event-Based Branch XL-form
010011 ///// ///// ///// 00000 10010/	III	XXXX			rfid	PPC	P		1086	Return From Interrupt Doubleword XL-form
010011 ///// ///// ///// 00010 10010/	III	XXXX			rfscv	v3.0	P		1085	Return From System Call Vectored XL-form
011110 .... .1000.	I	..XX			rldcl[.]	PPC		SR	111	Rotate Left Doubleword then Clear Left MDS-form
011110 .... .1001.	I	..XX			rldcr[.]	PPC		SR	112	Rotate Left Doubleword then Clear Right MDS-form
011110 .... .010..	I	..XX			rldic[.]	PPC		SR	111	Rotate Left Doubleword Immediate then Clear MD-form
011110 .... .000..	I	..XX			rldicl[.]	PPC		SR	110	Rotate Left Doubleword Immediate then Clear Left MD-form
011110 .... .001..	I	..XX			rldicr[.]	PPC		SR	110	Rotate Left Doubleword Immediate then Clear Right MD-form
011110 .... .011..	I	..XX			rldimi[.]	PPC		SR	112	Rotate Left Doubleword Immediate then Mask Insert MD-form
010100	I	XXXX			rlwimi[.]	P1		SR	109	Rotate Left Word Immediate then Mask Insert M-form
010101	I	XXXX			rlwinm[.]	P1		SR	108	Rotate Left Word Immediate then AND with Mask M-form
010111	I	XXXX			rlwnm[.]	P1		SR	109	Rotate Left Word then AND with Mask M-form
010001 ///// ///// ////. ..... .///1/	I	XXXX			sc	PPC			45	System Call SC-form
010001 ///// ///// ////. ..... .///01	I	XXXX			scv	v3.0			45	System Call Vectored SC-form
011111 ....// ///// 00100 00000/	I	XXXX			setb	v3.0			131	Set Boolean X-form
011111 .... ///// 01100 00000/	I	XXXX			setbc	v3.1			131	Set Boolean Condition X-form
011111 .... ///// 01101 00000/	I	XXXX			setbcr	v3.1			131	Set Boolean Condition Reverse X-form
011111 .... ///// 01110 00000/	I	XXXX			setnbc	v3.1			131	Set Negative Boolean Condition X-form
011111 .... ///// 01111 00000/	I	XXXX			setnbcr	v3.1			131	Set Negative Boolean Condition Reverse X-form
011111 ..... ///// 11110 100111	III	...X			slbfee.	v2.05	P	SR	1170	SLB Find Entry ESID X-form
011111 //... ///// ///// 01111 10010/	III	..XX			slbia	PPC	P		1164	SLB Invalidate All X-form
011111 ..... ////. ///// 11010 10010/	III	...X			slbiag	v3.0B	P		1167	SLB Invalidate All Global X-form
011111 ///// ///// 01101 10010/	III	...X			slbie	PPC	P		1160	SLB Invalidate Entry X-form
011111 ..... ///// 01110 10010/	III	...X			slbieg	v3.0	P		1162	SLB Invalidate Entry Global X-form
011111 ..... ///// 11100 10011/	III	...X			slbmfee	v2.00	P		1169	SLB Move From Entry ESID X-form
011111 ..... ///// 11010 10011/	III	...X			slbmfev	v2.00	P		1169	SLB Move From Entry VSID X-form
011111 ..... ///// 01100 10010/	III	...X			slbmte	v2.00	P		1168	SLB Move To Entry X-form
011111 ///// ///// ///// 01010 10010/	III	...X			slbsync	v3.0	P		1171	SLB Synchronize X-form
011111 .... 00000 11011.	I	..XX			sld[.]	PPC		SR	115	Shift Left Doubleword X-form
011111 .... 00000 11000.	I	XXXX			slw[.]	P1		SR	113	Shift Left Word X-form

Table H.1: Power ISA Instruction Set Sorted by Mnemonic (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
011111 .... 11000 11010.	I	..XX			srad[.]	PPC		SR	115	Shift Right Algebraic Doubleword X-form
011111 .... 11001 1101..	I	..XX			sradi[.]	PPC		SR	115	Shift Right Algebraic Doubleword Immediate XS-form
011111 .... 11000 11000.	I	XXXX			sraw[.]	P1		SR	114	Shift Right Algebraic Word X-form
011111 .... 11001 11000.	I	XXXX			srawi[.]	P1		SR	114	Shift Right Algebraic Word Immediate X-form
011111 .... 10000 11011.	I	..XX			srd[.]	PPC		SR	115	Shift Right Doubleword X-form
011111 .... 10000 11000.	I	XXXX			srw[.]	P1		SR	113	Shift Right Word X-form
100110	I	XXXX			stb	P1			57	Store Byte D-form
011111 .... 11110 10101/	III	..XX			stbcix	v2.05	HV		1098	Store Byte Caching Inhibited Indexed X-form
011111 .... 10101 101101	II	XXXX			stbcx.	v2.06			1018	Store Byte Conditional Indexed X-form
100111	I	XXXX			stbu	P1			57	Store Byte with Update D-form
011111 .... 00111 10111/	I	XXXX			stbux	P1			57	Store Byte with Update Indexed X-form
011111 .... 00110 10111/	I	XXXX			stbx	P1			57	Store Byte Indexed X-form
111110 00	I	..XX			std	PPC			60	Store Doubleword DS-form
011111 .... 10111 00110/	II	...X	AMO		stdat	v3.0			1013	Store Doubleword Atomic X-form
011111 .... 10100 10100/	I	..XX			stdbrx	v2.06			67	Store Doubleword Byte-Reverse Indexed X-form
011111 .... 11111 10101/	III	..XX			stdcix	v2.05	HV		1098	Store Doubleword Caching Inhibited Indexed X-form
011111 .... 00110 101101	II	..XX			stdcx.	PPC			1022	Store Doubleword Conditional Indexed X-form
111110 01	I	..XX			stdu	PPC			61	Store Doubleword with Update DS-form
011111 .... 00101 10101/	I	..XX			stdux	PPC			61	Store Doubleword with Update Indexed X-form
011111 .... 00100 10101/	I	..XX			stdx	PPC			60	Store Doubleword Indexed X-form
110110	I	.XXX			stfd	P1			156	Store Floating-Point Double D-form
111101 00	I	...X			stfdp	v2.05			159	Store Floating-Point Double Pair DS-form
011111 .... 11100 10111/	I	...X			stfdpx	v2.05			159	Store Floating-Point Double Pair Indexed X-form
110111	I	.XXX			stfdu	P1			156	Store Floating-Point Double with Update D-form
011111 .... 10111 10111/	I	.XXX			stfdux	P1			157	Store Floating-Point Double with Update Indexed X-form
011111 .... 10110 10111/	I	.XXX			stfdx	P1			156	Store Floating-Point Double Indexed X-form
011111 .... 11110 10111/	I	.XXX			stfiwx	PPC			157	Store Floating-Point as Integer Word Indexed X-form
110100	I	.XXX			stfs	P1			154	Store Floating-Point Single D-form
110101	I	.XXX			stfsu	P1			155	Store Floating-Point Single with Update D-form
011111 .... 10101 10111/	I	.XXX			stfsux	P1			155	Store Floating-Point Single with Update Indexed X-form
011111 .... 10100 10111/	I	.XXX			stfsx	P1			155	Store Floating-Point Single Indexed X-form
101100	I	XXXX			sth	P1			58	Store Halfword D-form
011111 .... 11100 10110/	I	XXXX			sthbrx	P1			66	Store Halfword Byte-Reverse Indexed X-form
011111 .... 11101 10101/	III	..XX			sthcix	v2.05	HV		1098	Store Halfword Caching Inhibited Indexed X-form
011111 .... 10110 101101	II	XXXX			sthcx.	v2.06			1019	Store Halfword Conditional Indexed X-form
101101	I	XXXX			sthu	P1			58	Store Halfword with Update D-form
011111 .... 01101 10111/	I	XXXX			sthux	P1			58	Store Halfword with Update Indexed X-form
011111 .... 01100 10111/	I	XXXX			sthx	P1			58	Store Halfword Indexed X-form
101111	I	...X			stmw	P1			68	Store Multiple Word D-form
010011 ///// ///// ///// 01011 10010/	III	..XX			stop	v3.0	P		1089	Stop XL-form
111110 10	I	..XX			stq	v2.03			64	Store Quadword DS-form
011111 .... 00101 101101	I	..XX			stqcx.	v2.07			1024	Store Quadword Conditional Indexed X-form
011111 .... 10110 10101/	I	...X			stswi	P1			71	Store String Word Immediate X-form
011111 .... 10100 10101/	I	...X			stswx	P1			71	Store String Word Indexed X-form
011111 .... 00100 00111/	I	..XX			stvebx	v2.03			263	Store Vector Element Byte Indexed X-form
011111 .... 00101 00111/	I	..XX			stvehx	v2.03			264	Store Vector Element Halfword Indexed X-form
011111 .... 00110 00111/	I	..XX			stvewx	v2.03			264	Store Vector Element Word Indexed X-form
011111 .... 00111 00111/	I	..XX			stvx	v2.03			265	Store Vector Indexed X-form

Table H.1: Power ISA Instruction Set Sorted by Mnemonic (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
011111 .... 01111 00111/	I	..XX			stvxl	v2.03			265	Store Vector Indexed Last X-form
100100	I	XXXX			stw	P1			59	Store Word D-form
011111 .... 10110 00110/	II	...X	AMO		stwat	v3.0			1013	Store Word Atomic X-form
011111 .... 10100 10110/	I	XXXX			stwbrx	P1			66	Store Word Byte-Reverse Indexed X-form
011111 .... 11100 10101/	III	..XX			stwcix	v2.05	HV		1098	Store Word Caching Inhibited Indexed X-form
011111 .... 00100 101101	II	XXXX			stwcx.	PPC			1020	Store Word Conditional Indexed X-form
100101	I	XXXX			stwu	P1			59	Store Word with Update D-form
011111 .... 00101 10111/	I	XXXX			stwux	P1			59	Store Word with Update Indexed X-form
011111 .... 00100 10111/	I	XXXX			stwx	P1			59	Store Word Indexed X-form
111101 10	I	..XX			stxsd	v3.0			569	Store VSX Scalar Doubleword DS-form
011111 .... 10110 01100.	I	..XX			stxsdx	v2.06			570	Store VSX Scalar Doubleword Indexed X-form
011111 .... 11100 01101.	I	..XX			stxsibx	v3.0			571	Store VSX Scalar as Integer Byte Indexed X-form
011111 .... 11101 01101.	I	..XX			stxsihx	v3.0			571	Store VSX Scalar as Integer Halfword Indexed X-form
011111 .... 00100 01100.	I	..XX			stxsiwx	v2.07			572	Store VSX Scalar as Integer Word Indexed X-form
111101 11	I	..XX			stxssp	v3.0			573	Store VSX Scalar Single-Precision DS-form
011111 .... 10100 01100.	I	..XX			stxsspx	v2.07			574	Store VSX Scalar Single-Precision Indexed X-form
111101 101	I	..XX			stxv	v3.0			591	Store VSX Vector DQ-form
011111 .... 11111 01100.	I	..XX			stxvb16x	v3.0			592	Store VSX Vector Byte*16 Indexed X-form
011111 .... 11110 01100.	I	..XX			stxvd2x	v2.06			593	Store VSX Vector Doubleword*2 Indexed X-form
011111 .... 11101 01100.	I	..XX			stxvh8x	v3.0			594	Store VSX Vector Halfword*8 Indexed X-form
011111 .... 01100 01101.	I	..XX			stxvl	v3.0			600	Store VSX Vector with Length X-form
011111 .... 01101 01101.	I	..XX			stxvll	v3.0			602	Store VSX Vector with Length Left-justified X-form
000110 0001	I	..XX			stxvp	v3.1			605	Store VSX Vector Paired DQ-form
011111 .... 01110 01101/	I	..XX			stxvpx	v3.1			606	Store VSX Vector Paired Indexed X-form
011111 .... 00100 01101.	I	..XX			stxvrbx	v3.1			598	Store VSX Vector Rightmost Byte Indexed X-form
011111 .... 00111 01101.	I	..XX			stxvrdx	v3.1			598	Store VSX Vector Rightmost Doubleword Indexed X-form
011111 .... 00101 01101.	I	..XX			stxvrhx	v3.1			599	Store VSX Vector Rightmost Halfword Indexed X-form
011111 .... 00110 01101.	I	..XX			stxvrwx	v3.1			599	Store VSX Vector Rightmost Word Indexed X-form
011111 .... 11100 01100.	I	..XX			stxvw4x	v2.06			595	Store VSX Vector Word*4 Indexed X-form
011111 .... 01100 01100.	I	..XX			stxvx	v3.0			596	Store VSX Vector Indexed X-form
011111 .... 00001 01000.	I	XXXX			subf[.]	PPC		SR	75	Subtract From XO-form
011111 .... 00000 01000.	I	XXXX			subfc[.]	P1		SR	76	Subtract From Carrying XO-form
011111 .... 10000 01000.	I	..XX			subfco[.]	P1		SR	76	Subtract From Carrying & record OV XO-form
011111 .... 00100 01000.	I	XXXX			subfe[.]	P1		SR	76	Subtract From Extended XO-form
011111 .... 10100 01000.	I	..XX			subfeo[.]	P1		SR	76	Subtract From Extended & record OV XO-form
001000	I	XXXX			subfic	P1		SR	75	Subtract From Immediate Carrying D-form
011111 .... ///// 00111 01000.	I	XXXX			subfme[.]	P1		SR	77	Subtract From Minus One Extended XO-form
011111 .... ///// 10111 01000.	I	..XX			subfmeo[.]	P1		SR	77	Subtract From Minus One Extended & record OV XO-form
011111 .... 10001 01000.	I	..XX			subfo[.]	PPC		SR	75	Subtract From & record OV XO-form
011111 .... ///// 00110 01000.	I	XXXX			subfze[.]	P1		SR	77	Subtract From Zero Extended XO-form
011111 .... ///// 10110 01000.	I	..XX			subfzeo[.]	P1		SR	77	Subtract From Zero Extended & record OV XO-form
011111 //... ///.. ///// 10010 10110/	II	XXXX			sync	P1			1025	Synchronize X-form
011111 .... 00010 00100/	I	..XX			td	PPC			95	Trap Doubleword X-form
000010	I	..XX			tdi	PPC			95	Trap Doubleword Immediate D-form
011111 ..... /.... ..... 01001 10010/	III	...X			tlbie	P1	HV	64	1174	TLB Invalidate Entry X-form
011111 ..... /.... ..... 01000 10010/	III	..XX			tlbiel	v2.03	P	64	1181	TLB Invalidate Entry Local X-form
011111 ///// ///// ///// 10001 10110/	III	...X			tlbsync	PPC	HV/P		1186	TLB Synchronize X-form
011111 .... 00000 00100/	I	XXXX			tw	P1			94	Trap Word X-form

Table H.1: Power ISA Instruction Set Sorted by Mnemonic (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
000011	I	XXXX			twi	P1			94	Trap Word Immediate D-form
010011 ///// ///// ///// 01001 10010/	III	....			urfid	v3.0C	UV		1087	Ultravisor Return From Interrupt Doubleword XL-form
000100 .... 10000 000011	I	..XX			vabsdub	v3.0			368	Vector Absolute Difference Unsigned Byte VX-form
000100 .... 10001 000011	I	..XX			vabsduh	v3.0			368	Vector Absolute Difference Unsigned Halfword VX-form
000100 .... 10010 000011	I	..XX			vabsduw	v3.0			369	Vector Absolute Difference Unsigned Word VX-form
000100 .... 00101 000000	I	..XX			vaddcuq	v2.07			320	Vector Add & write Carry-out Unsigned Quadword VX-form
000100 .... 00110 000000	I	..XX			vaddcuw	v2.03			313	Vector Add & write Carry-out Unsigned Word VX-form
000100 111101	I	..XX			vaddecuq	v2.07			320	Vector Add Extended & write Carry-out Unsigned Quadword VA-form
000100 111100	I	..XX			vaddeuqm	v2.07			319	Vector Add Extended Unsigned Quadword Modulo VA-form
000100 .... 00000 001010	I	..XX			vaddfp	v2.03			411	Vector Add Floating-Point VX-form
000100 .... 01100 000000	I	..XX			vaddsbs	v2.03			313	Vector Add Signed Byte Saturate VX-form
000100 .... 01101 000000	I	..XX			vaddshs	v2.03			314	Vector Add Signed Halfword Saturate VX-form
000100 .... 01110 000000	I	..XX			vaddsws	v2.03			314	Vector Add Signed Word Saturate VX-form
000100 .... 00000 000000	I	..XX			vaddubm	v2.03			315	Vector Add Unsigned Byte Modulo VX-form
000100 .... 01000 000000	I	..XX			vaddubs	v2.03			317	Vector Add Unsigned Byte Saturate VX-form
000100 .... 00011 000000	I	..XX			vaddudm	v2.07			316	Vector Add Unsigned Doubleword Modulo VX-form
000100 .... 00001 000000	I	..XX			vadduhm	v2.03			315	Vector Add Unsigned Halfword Modulo VX-form
000100 .... 01001 000000	I	..XX			vadduhs	v2.03			317	Vector Add Unsigned Halfword Saturate VX-form
000100 .... 00100 000000	I	..XX			vadduqm	v2.07			319	Vector Add Unsigned Quadword Modulo VX-form
000100 .... 00010 000000	I	..XX			vadduwm	v2.03			316	Vector Add Unsigned Word Modulo VX-form
000100 .... 01010 000000	I	..XX			vadduws	v2.03			318	Vector Add Unsigned Word Saturate VX-form
000100 .... 10000 000100	I	..XX			vand	v2.03			392	Vector Logical AND VX-form
000100 .... 10001 000100	I	..XX			vandc	v2.03			392	Vector Logical AND with Complement VX-form
000100 .... 10100 000010	I	..XX			vavgsb	v2.03			365	Vector Average Signed Byte VX-form
000100 .... 10101 000010	I	..XX			vavgsh	v2.03			366	Vector Average Signed Halfword VX-form
000100 .... 10110 000010	I	..XX			vavgsw	v2.03			367	Vector Average Signed Word VX-form
000100 .... 10000 000010	I	..XX			vavgub	v2.03			365	Vector Average Unsigned Byte VX-form
000100 .... 10001 000010	I	..XX			vavguh	v2.03			366	Vector Average Unsigned Halfword VX-form
000100 .... 10010 000010	I	..XX			vavguw	v2.03			367	Vector Average Unsigned Word VX-form
000100 .... 10111 001100	I	..XX			vbpermd	v3.0			449	Vector Bit Permute Doubleword VX-form
000100 .... 10101 001100	I	..XX			vbpermq	v2.07			450	Vector Bit Permute Quadword VX-form
000100 .... 01101 001010	I	..XX			vcfsx	v2.03			415	Vector Convert with round to nearest From Signed Word to floating-point format VX-form
000100 .... 10101 001101	I	..XX			vcfuged	v3.1			444	Vector Centrifuge Doubleword VX-form
000100 .... 01100 001010	I	..XX			vcfux	v2.03			415	Vector Convert with round to nearest From Unsigned Word to floating-point format VX-form
000100 .... 10100 001000	I	..XX			vcipher	v2.07			424	Vector AES Cipher VX-form
000100 .... 10100 001001	I	..XX			vcipherlast	v2.07			424	Vector AES Cipher Last VX-form
000100 .... 00110 001101	I	..XX			vclrlb	v3.1			464	Vector Clear Leftmost Bytes VX-form
000100 .... 00111 001101	I	..XX			vclrrb	v3.1			464	Vector Clear Rightmost Bytes VX-form
000100 ..... ///// 11100 000010	I	..XX			vclzb	v2.07			435	Vector Count Leading Zeros Byte VX-form
000100 ..... ///// 11111 000010	I	..XX			vclzd	v2.07			437	Vector Count Leading Zeros Doubleword VX-form
000100 .... 11110 000100	I	..XX			vclzdm	v3.1			437	Vector Count Leading Zeros Doubleword under bit Mask VX-form
000100 ..... ///// 11101 000010	I	..XX			vclzh	v2.07			435	Vector Count Leading Zeros Halfword VX-form

Table H.1: Power ISA Instruction Set Sorted by Mnemonic (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
000100 ..... 00000 ..... 11000 000010	I	..XX			vclzlsbb	v3.0			441	Vector Count Leading Zero Least-Significant Bits Byte VX-form
000100 ..... ///// 11110 000010	I	..XX			vclzw	v2.07			436	Vector Count Leading Zeros Word VX-form
000100 .... .1111 000110	I	..XX			vcmpbfp[.]	v2.03			418	Vector Compare Bounds Floating-Point VC-form
000100 .... .0011 000110	I	..XX			vcmpeqfp[.]	v2.03			419	Vector Compare Equal Floating-Point VC-form
000100 .... .0000 000110	I	..XX			vcmpequb[.]	v2.03			378	Vector Compare Equal Unsigned Byte VC-form
000100 .... .0011 000111	I	..XX			vcmpequd[.]	v2.07			381	Vector Compare Equal Unsigned Doubleword VC-form
000100 .... .0001 000110	I	..XX			vcmpequh[.]	v2.03			379	Vector Compare Equal Unsigned Halfword VC-form
000100 .... .0111 000111	I	..XX			vcmpequq[.]	v3.1			382	Vector Compare Equal Quadword VC-form
000100 .... .0010 000110	I	..XX			vcmpequw[.]	v2.03			380	Vector Compare Equal Unsigned Word VC-form
000100 .... .0111 000110	I	..XX			vcmpgefp[.]	v2.03			419	Vector Compare Greater Than or Equal Floating-Point VC-form
000100 .... .1011 000110	I	..XX			vcmpgtfp[.]	v2.03			420	Vector Compare Greater Than Floating-Point VC-form
000100 .... .1100 000110	I	..XX			vcmpgtsb[.]	v2.03			383	Vector Compare Greater Than Signed Byte VC-form
000100 .... .1111 000111	I	..XX			vcmpgtsd[.]	v2.07			386	Vector Compare Greater Than Signed Doubleword VC-form
000100 .... .1101 000110	I	..XX			vcmpgtsh[.]	v2.03			384	Vector Compare Greater Than Signed Halfword VC-form
000100 .... .1110 000111	I	..XX			vcmpgtsq[.]	v3.1			387	Vector Compare Greater Than Signed Quadword VC-form
000100 .... .1110 000110	I	..XX			vcmpgtsw[.]	v2.03			385	Vector Compare Greater Than Signed Word VC-form
000100 .... .1000 000110	I	..XX			vcmpgtub[.]	v2.03			383	Vector Compare Greater Than Unsigned Byte VC-form
000100 .... .1011 000111	I	..XX			vcmpgtud[.]	v2.07			386	Vector Compare Greater Than Unsigned Doubleword VC-form
000100 .... .1001 000110	I	..XX			vcmpgtuh[.]	v2.03			384	Vector Compare Greater Than Unsigned Halfword VC-form
000100 .... .1010 000111	I	..XX			vcmpgtuq[.]	v3.1			387	Vector Compare Greater Than Unsigned Quadword VC-form
000100 .... .1010 000110	I	..XX			vcmpgtuw[.]	v2.03			385	Vector Compare Greater Than Unsigned Word VC-form
000100 .... .0000 000111	I	..XX			vcmpneb[.]	v3.0			388	Vector Compare Not Equal Byte VC-form
000100 .... .0001 000111	I	..XX			vcmpneh[.]	v3.0			389	Vector Compare Not Equal Halfword VC-form
000100 .... .0010 000111	I	..XX			vcmpnew[.]	v3.0			390	Vector Compare Not Equal Word VC-form
000100 .... .0100 000111	I	..XX			vcmpnezb[.]	v3.0			388	Vector Compare Not Equal or Zero Byte VC-form
000100 .... .0101 000111	I	..XX			vcmpnezh[.]	v3.0			389	Vector Compare Not Equal or Zero Halfword VC-form
000100 .... .0110 000111	I	..XX			vcmpnezw[.]	v3.0			390	Vector Compare Not Equal or Zero Word VC-form
000100 .../// 00101 000001	I	..XX			vcmpsq	v3.1			391	Vector Compare Signed Quadword VX-form
000100 .../// 00100 000001	I	..XX			vcmpuq	v3.1			391	Vector Compare Unsigned Quadword VX-form
000100 ..... 1100. ..... 11001 000010	I	..XX			vcntmbb	v3.1			457	Vector Count Mask Bits Byte VX-form
000100 ..... 1111. ..... 11001 000010	I	..XX			vcntmbd	v3.1			458	Vector Count Mask Bits Doubleword VX-form
000100 ..... 1101. ..... 11001 000010	I	..XX			vcntmbh	v3.1			457	Vector Count Mask Bits Halfword VX-form
000100 ..... 1110. ..... 11001 000010	I	..XX			vcntmbw	v3.1			458	Vector Count Mask Bits Word VX-form
000100 .... 01111 001010	I	..XX			vctsxs	v2.03			414	Vector Convert with round to zero from floating-point To Signed Word format Saturate VX-form
000100 .... 01110 001010	I	..XX			vctuxs	v2.03			414	Vector Convert with round to zero from floating-point To Unsigned Word format Saturate VX-form
000100 ..... 11100 ..... 11000 000010	I	..XX			vctzb	v3.0			438	Vector Count Trailing Zeros Byte VX-form
000100 ..... 11111 ..... 11000 000010	I	..XX			vctzd	v3.0			440	Vector Count Trailing Zeros Doubleword VX-form
000100 .... 11111 000100	I	..XX			vctzdm	v3.1			440	Vector Count Trailing Zeros Doubleword under bit Mask VX-form
000100 ..... 11101 ..... 11000 000010	I	..XX			vctzh	v3.0			438	Vector Count Trailing Zeros Halfword VX-form

Table H.1: Power ISA Instruction Set Sorted by Mnemonic (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
000100 ..... 00001 ..... 11000 000010	I	..XX			vctzlsbb	v3.0			441	Vector Count Trailing Zero Least-Significant Bits Byte VX-form
000100 ..... 11110 ..... 11000 000010	I	..XX			vctzw	v3.0			439	Vector Count Trailing Zeros Word VX-form
000100 .... 01111 001011	I	..XX			vdivesd	v3.1			351	Vector Divide Extended Signed Doubleword VX-form
000100 .... 01100 001011	I	..XX			vdivesq	v3.1			353	Vector Divide Extended Signed Quadword VX-form
000100 .... 01110 001011	I	..XX			vdivesw	v3.1			349	Vector Divide Extended Signed Word VX-form
000100 .... 01011 001011	I	..XX			vdiveud	v3.1			351	Vector Divide Extended Unsigned Doubleword VX-form
000100 .... 01000 001011	I	..XX			vdiveuq	v3.1			353	Vector Divide Extended Unsigned Quadword VX-form
000100 .... 01010 001011	I	..XX			vdiveuw	v3.1			349	Vector Divide Extended Unsigned Word VX-form
000100 .... 00111 001011	I	..XX			vdivsd	v3.1			350	Vector Divide Signed Doubleword VX-form
000100 .... 00100 001011	I	..XX			vdivsq	v3.1			352	Vector Divide Signed Quadword VX-form
000100 .... 00110 001011	I	..XX			vdivsw	v3.1			348	Vector Divide Signed Word VX-form
000100 .... 00011 001011	I	..XX			vdivud	v3.1			350	Vector Divide Unsigned Doubleword VX-form
000100 .... 00000 001011	I	..XX			vdivuq	v3.1			352	Vector Divide Unsigned Quadword VX-form
000100 .... 00010 001011	I	..XX			vdivuw	v3.1			348	Vector Divide Unsigned Word VX-form
000100 .... 11010 000100	I	..XX			veqv	v2.07			393	Vector Logical Equivalence VX-form
000100 ..... 00000 ..... 11001 000010	I	..XX			vexpandbm	v3.1			454	Vector Expand Byte Mask VX-form
000100 ..... 00011 ..... 11001 000010	I	..XX			vexpanddm	v3.1			455	Vector Expand Doubleword Mask VX-form
000100 ..... 00001 ..... 11001 000010	I	..XX			vexpandhm	v3.1			454	Vector Expand Halfword Mask VX-form
000100 ..... 00100 ..... 11001 000010	I	..XX			vexpandqm	v3.1			456	Vector Expand Quadword Mask VX-form
000100 ..... 00010 ..... 11001 000010	I	..XX			vexpandwm	v3.1			455	Vector Expand Word Mask VX-form
000100 ..... ///// 00110 001010	I	..XX			vexptefp	v2.03			421	Vector 2 Raised to the Exponent Estimate Floating-Point VX-form
000100 011110	I	..XX			vextddvlx	v3.1			302	Vector Extract Double Doubleword to VSR using GPR-specified Left-Index VA-form
000100 011111	I	..XX			vextddvrx	v3.1			302	Vector Extract Double Doubleword to VSR using GPR-specified Right-Index VA-form
000100 011000	I	..XX			vextdubvlx	v3.1			299	Vector Extract Double Unsigned Byte to VSR using GPR-specified Left-Index VA-form
000100 011001	I	..XX			vextdubvrx	v3.1			299	Vector Extract Double Unsigned Byte to VSR using GPR-specified Right-Index VA-form
000100 011010	I	..XX			vextduhvlx	v3.1			300	Vector Extract Double Unsigned Halfword to VSR using GPR-specified Left-Index VA-form
000100 011011	I	..XX			vextduhvrx	v3.1			300	Vector Extract Double Unsigned Halfword to VSR using GPR-specified Right-Index VA-form
000100 011100	I	..XX			vextduwvlx	v3.1			301	Vector Extract Double Unsigned Word to VSR using GPR-specified Left-Index VA-form
000100 011101	I	..XX			vextduwvrx	v3.1			301	Vector Extract Double Unsigned Word to VSR using GPR-specified Right-Index VA-form
000100 ..... 01000 ..... 11001 000010	I	..XX			vextractbm	v3.1			459	Vector Extract Byte Mask VX-form
000100 ..... /.... ..... 01011 001101	I	..XX			vextractd	v3.0			295	Vector Extract Doubleword to VSR using immediate-specified index VX-form
000100 ..... 01011 ..... 11001 000010	I	..XX			vextractdm	v3.1			460	Vector Extract Doubleword Mask VX-form
000100 ..... 01001 ..... 11001 000010	I	..XX			vextracthm	v3.1			459	Vector Extract Halfword Mask VX-form
000100 ..... 01100 ..... 11001 000010	I	..XX			vextractqm	v3.1			461	Vector Extract Quadword Mask VX-form
000100 ..... /.... ..... 01000 001101	I	..XX			vextractub	v3.0			294	Vector Extract Unsigned Byte to VSR using immediate-specified index VX-form
000100 ..... /.... ..... 01001 001101	I	..XX			vextractuh	v3.0			294	Vector Extract Unsigned Halfword to VSR using immediate-specified index VX-form

Table H.1: Power ISA Instruction Set Sorted by Mnemonic (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
000100 ..... /.... ..... 01010 001101	I	..XX			vextractuw	v3.0			295	Vector Extract Unsigned Word to VSR using immediate-specified index VX-form
000100 ..... 01010 ..... 11001 000010	I	..XX			vextractwm	v3.1			460	Vector Extract Word Mask VX-form
000100 ..... 11000 ..... 11000 000010	I	..XX			vextsb2d	v3.0			363	Vector Extend Sign Byte To Doubleword VX-form
000100 ..... 10000 ..... 11000 000010	I	..XX			vextsb2w	v3.0			362	Vector Extend Sign Byte To Word VX-form
000100 ..... 11011 ..... 11000 000010	I	..XX			vextsd2q	v3.1			364	Vector Extend Sign Doubleword to Quadword VX-form
000100 ..... 11001 ..... 11000 000010	I	..XX			vextsh2d	v3.0			363	Vector Extend Sign Halfword To Doubleword VX-form
000100 ..... 10001 ..... 11000 000010	I	..XX			vextsh2w	v3.0			362	Vector Extend Sign Halfword To Word VX-form
000100 ..... 11010 ..... 11000 000010	I	..XX			vextsw2d	v3.0			364	Vector Extend Sign Word To Doubleword VX-form
000100 .... 11000 001101	I	..XX			vextublx	v3.0			296	Vector Extract Unsigned Byte to GPR using GPR-specified Left-Index VX-form
000100 .... 11100 001101	I	..XX			vextubrx	v3.0			296	Vector Extract Unsigned Byte to GPR using GPR-specified Right-Index VX-form
000100 .... 11001 001101	I	..XX			vextuhlx	v3.0			297	Vector Extract Unsigned Halfword to GPR using GPR-specified Left-Index VX-form
000100 .... 11101 001101	I	..XX			vextuhrx	v3.0			297	Vector Extract Unsigned Halfword to GPR using GPR-specified Right-Index VX-form
000100 .... 11010 001101	I	..XX			vextuwlx	v3.0			298	Vector Extract Unsigned Word to GPR using GPR-specified Left-Index VX-form
000100 .... 11110 001101	I	..XX			vextuwrx	v3.0			298	Vector Extract Unsigned Word to GPR using GPR-specified Right-Index VX-form
000100 ..... ///// 10100 001100	I	..XX			vgbbd	v2.07			433	Vector Gather Bits by Bytes by Doubleword VX-form
000100 ..... //... ..... 10011 001100	I	..XX			vgnb	v3.1			434	Vector Gather every Nth Bit VX-form
000100 .... 01000 001111	I	..XX			vinsblx	v3.1			305	Vector Insert Byte from GPR using GPR-specified Left-Index VX-form
000100 .... 01100 001111	I	..XX			vinsbrx	v3.1			305	Vector Insert Byte from GPR using GPR-specified Right-Index VX-form
000100 .... 00000 001111	I	..XX			vinsbvlx	v3.1			310	Vector Insert Byte from VSR using GPR-specified Left-Index VX-form
000100 .... 00100 001111	I	..XX			vinsbvrx	v3.1			310	Vector Insert Byte from VSR using GPR-specified Right-Index VX-form
000100 ..... /.... ..... 00111 001111	I	..XX			vinsd	v3.1			309	Vector Insert Doubleword from GPR using immediate-specified index VX-form
000100 .... 01011 001111	I	..XX			vinsdlx	v3.1			308	Vector Insert Doubleword from GPR using GPR-specified Left-Index VX-form
000100 .... 01111 001111	I	..XX			vinsdrx	v3.1			308	Vector Insert Doubleword from GPR using GPR-specified Right-Index VX-form
000100 ..... /.... ..... 01100 001101	I	..XX			vinsertb	v3.0			303	Vector Insert Byte from VSR using immediate-specified index VX-form
000100 ..... /.... ..... 01111 001101	I	..XX			vinsertd	v3.0			304	Vector Insert Doubleword from VSR using immediate-specified index VX-form
000100 ..... /.... ..... 01101 001101	I	..XX			vinserth	v3.0			303	Vector Insert Halfword from VSR using immediate-specified index VX-form
000100 ..... /.... ..... 01110 001101	I	..XX			vinsertw	v3.0			304	Vector Insert Word from VSR using immediate-specified index VX-form
000100 .... 01001 001111	I	..XX			vinshlx	v3.1			306	Vector Insert Halfword from GPR using GPR-specified Left-Index VX-form
000100 .... 01101 001111	I	..XX			vinshrx	v3.1			306	Vector Insert Halfword from GPR using GPR-specified Right-Index VX-form

Table H.1: Power ISA Instruction Set Sorted by Mnemonic (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
000100 .... 00001 001111	I	..XX			vinshvlx	v3.1			311	Vector Insert Halfword from VSR using GPR-specified Left-Index VX-form
000100 .... 00101 001111	I	..XX			vinshvrx	v3.1			311	Vector Insert Halfword from VSR using GPR-specified Right-Index VX-form
000100 ..... /.... ..... 00011 001111	I	..XX			vinsw	v3.1			309	Vector Insert Word from GPR using immediate-specified index VX-form
000100 .... 01010 001111	I	..XX			vinswlx	v3.1			307	Vector Insert Word from GPR using GPR-specified Left-Index VX-form
000100 .... 01110 001111	I	..XX			vinswrx	v3.1			307	Vector Insert Word from GPR using GPR-specified Right-Index VX-form
000100 .... 00010 001111	I	..XX			vinswvlx	v3.1			312	Vector Insert Word from VSR using GPR-specified Left-Index VX-form
000100 .... 00110 001111	I	..XX			vinswvrx	v3.1			312	Vector Insert Word from VSR using GPR-specified Right-Index VX-form
000100 ..... ///// 00111 001010	I	..XX			vlogefp	v2.03			422	Vector Log Base 2 Estimate Floating-Point VX-form
000100 101110	I	..XX			vmaddfp	v2.03			412	Vector Multiply-Add Floating-Point VA-form
000100 .... 10000 001010	I	..XX			vmaxfp	v2.03			413	Vector Maximum Floating-Point VX-form
000100 .... 00100 000010	I	..XX			vmaxsb	v2.03			370	Vector Maximum Signed Byte VX-form
000100 .... 00111 000010	I	..XX			vmaxsd	v2.07			373	Vector Maximum Signed Doubleword VX-form
000100 .... 00101 000010	I	..XX			vmaxsh	v2.03			371	Vector Maximum Signed Halfword VX-form
000100 .... 00110 000010	I	..XX			vmaxsw	v2.03			372	Vector Maximum Signed Word VX-form
000100 .... 00000 000010	I	..XX			vmaxub	v2.03			370	Vector Maximum Unsigned Byte VX-form
000100 .... 00011 000010	I	..XX			vmaxud	v2.07			373	Vector Maximum Unsigned Doubleword VX-form
000100 .... 00001 000010	I	..XX			vmaxuh	v2.03			371	Vector Maximum Unsigned Halfword VX-form
000100 .... 00010 000010	I	..XX			vmaxuw	v2.03			372	Vector Maximum Unsigned Word VX-form
000100 100000	I	..XX			vmhaddshs	v2.03			341	Vector Multiply-High-Add Signed Halfword Saturate VA-form
000100 100001	I	..XX			vmhraddshs	v2.03			341	Vector Multiply-High-Round-Add Signed Halfword Saturate VA-form
000100 .... 10001 001010	I	..XX			vminfp	v2.03			413	Vector Minimum Floating-Point VX-form
000100 .... 01100 000010	I	..XX			vminsb	v2.03			374	Vector Minimum Signed Byte VX-form
000100 .... 01111 000010	I	..XX			vminsd	v2.07			377	Vector Minimum Signed Doubleword VX-form
000100 .... 01101 000010	I	..XX			vminsh	v2.03			375	Vector Minimum Signed Halfword VX-form
000100 .... 01110 000010	I	..XX			vminsw	v2.03			376	Vector Minimum Signed Word VX-form
000100 .... 01000 000010	I	..XX			vminub	v2.03			374	Vector Minimum Unsigned Byte VX-form
000100 .... 01011 000010	I	..XX			vminud	v2.07			377	Vector Minimum Unsigned Doubleword VX-form
000100 .... 01001 000010	I	..XX			vminuh	v2.03			375	Vector Minimum Unsigned Halfword VX-form
000100 .... 01010 000010	I	..XX			vminuw	v2.03			376	Vector Minimum Unsigned Word VX-form
000100 100010	I	..XX			vmladduhm	v2.03			342	Vector Multiply-Low-Add Unsigned Halfword Modulo VA-form
000100 .... 11111 001011	I	..XX			vmodsd	v3.1			355	Vector Modulo Signed Doubleword VX-form
000100 .... 11100 001011	I	..XX			vmodsq	v3.1			356	Vector Modulo Signed Quadword VX-form
000100 .... 11110 001011	I	..XX			vmodsw	v3.1			354	Vector Modulo Signed Word VX-form
000100 .... 11011 001011	I	..XX			vmodud	v3.1			355	Vector Modulo Unsigned Doubleword VX-form
000100 .... 11000 001011	I	..XX			vmoduq	v3.1			356	Vector Modulo Unsigned Quadword VX-form
000100 .... 11010 001011	I	..XX			vmoduw	v3.1			354	Vector Modulo Unsigned Word VX-form
000100 .... 11110 001100	I	..XX			vmrgew	v2.07			282	Vector Merge Even Word VX-form
000100 .... 00000 001100	I	..XX			vmrghb	v2.03			279	Vector Merge High Byte VX-form
000100 .... 00001 001100	I	..XX			vmrghh	v2.03			280	Vector Merge High Halfword VX-form

Table H.1: Power ISA Instruction Set Sorted by Mnemonic (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
000100 .... 00010 001100	I	..XX			vmrghw	v2.03			281	Vector Merge High Word VX-form
000100 .... 00100 001100	I	..XX			vmrglb	v2.03			279	Vector Merge Low Byte VX-form
000100 .... 00101 001100	I	..XX			vmrglh	v2.03			280	Vector Merge Low Halfword VX-form
000100 .... 00110 001100	I	..XX			vmrglw	v2.03			281	Vector Merge Low Word VX-form
000100 .... 11010 001100	I	..XX			vmrgow	v2.07			282	Vector Merge Odd Word VX-form
000100 010111	I	..XX			vmsumcud	v3.1			347	Vector Multiply-Sum & write Carry-out Unsigned Doubleword VA-form
000100 100101	I	..XX			vmsummbm	v2.03			343	Vector Multiply-Sum Mixed Byte Modulo VA-form
000100 101000	I	..XX			vmsumshm	v2.03			343	Vector Multiply-Sum Signed Halfword Modulo VA-form
000100 101001	I	..XX			vmsumshs	v2.03			344	Vector Multiply-Sum Signed Halfword Saturate VA-form
000100 100100	I	..XX			vmsumubm	v2.03			342	Vector Multiply-Sum Unsigned Byte Modulo VA-form
000100 100011	I	..XX			vmsumudm	v3.0B			346	Vector Multiply-Sum Unsigned Doubleword Modulo VA-form
000100 100110	I	..XX			vmsumuhm	v2.03			344	Vector Multiply-Sum Unsigned Halfword Modulo VA-form
000100 100111	I	..XX			vmsumuhs	v2.03			345	Vector Multiply-Sum Unsigned Halfword Saturate VA-form
000100 .... ///// 00000 000001	I	..XX			vmul10cuq	v3.0			474	Vector Multiply-by-10 & write Carry-out Unsigned Quadword VX-form
000100 .... 00001 000001	I	..XX			vmul10ecuq	v3.0			475	Vector Multiply-by-10 Extended & write Carry-out Unsigned Quadword VX-form
000100 .... 01001 000001	I	..XX			vmul10euq	v3.0			475	Vector Multiply-by-10 Extended Unsigned Quadword VX-form
000100 .... ///// 01000 000001	I	..XX			vmul10uq	v3.0			474	Vector Multiply-by-10 Unsigned Quadword VX-form
000100 .... 01100 001000	I	..XX			vmulesb	v2.03			329	Vector Multiply Even Signed Byte VX-form
000100 .... 01111 001000	I	..XX			vmulesd	v3.1			336	Vector Multiply Even Signed Doubleword VX-form
000100 .... 01101 001000	I	..XX			vmulesh	v2.03			331	Vector Multiply Even Signed Halfword VX-form
000100 .... 01110 001000	I	..XX			vmulesw	v2.07			333	Vector Multiply Even Signed Word VX-form
000100 .... 01000 001000	I	..XX			vmuleub	v2.03			330	Vector Multiply Even Unsigned Byte VX-form
000100 .... 01011 001000	I	..XX			vmuleud	v3.1			335	Vector Multiply Even Unsigned Doubleword VX-form
000100 .... 01001 001000	I	..XX			vmuleuh	v2.03			332	Vector Multiply Even Unsigned Halfword VX-form
000100 .... 01010 001000	I	..XX			vmuleuw	v2.07			334	Vector Multiply Even Unsigned Word VX-form
000100 .... 01111 001001	I	..XX			vmulhsd	v3.1			339	Vector Multiply High Signed Doubleword VX-form
000100 .... 01110 001001	I	..XX			vmulhsw	v3.1			338	Vector Multiply High Signed Word VX-form
000100 .... 01011 001001	I	..XX			vmulhud	v3.1			339	Vector Multiply High Unsigned Doubleword VX-form
000100 .... 01010 001001	I	..XX			vmulhuw	v3.1			338	Vector Multiply High Unsigned Word VX-form
000100 .... 00111 001001	I	..XX			vmulld	v3.1			340	Vector Multiply Low Doubleword VX-form
000100 .... 00100 001000	I	..XX			vmulosb	v2.03			329	Vector Multiply Odd Signed Byte VX-form
000100 .... 00111 001000	I	..XX			vmulosd	v3.1			336	Vector Multiply Odd Signed Doubleword VX-form
000100 .... 00101 001000	I	..XX			vmulosh	v2.03			331	Vector Multiply Odd Signed Halfword VX-form
000100 .... 00110 001000	I	..XX			vmulosw	v2.07			333	Vector Multiply Odd Signed Word VX-form
000100 .... 00000 001000	I	..XX			vmuloub	v2.03			330	Vector Multiply Odd Unsigned Byte VX-form
000100 .... 00011 001000	I	..XX			vmuloud	v3.1			335	Vector Multiply Odd Unsigned Doubleword VX-form
000100 .... 00001 001000	I	..XX			vmulouh	v2.03			332	Vector Multiply Odd Unsigned Halfword VX-form
000100 .... 00010 001000	I	..XX			vmulouw	v2.07			334	Vector Multiply Odd Unsigned Word VX-form
000100 .... 00010 001001	I	..XX			vmuluwm	v2.07			337	Vector Multiply Unsigned Word Modulo VX-form
000100 .... 10110 000100	I	..XX			vnand	v2.07			393	Vector Logical NAND VX-form
000100 .... 10101 001000	I	..XX			vncipher	v2.07			425	Vector AES Inverse Cipher VX-form
000100 .... 10101 001001	I	..XX			vncipherlast	v2.07			425	Vector AES Inverse Cipher Last VX-form
000100 ..... 00111 ..... 11000 000010	I	..XX			vnegd	v3.0			361	Vector Negate Doubleword VX-form

Table H.1: Power ISA Instruction Set Sorted by Mnemonic (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
000100 ..... 00110 ..... 11000 000010	I	..XX			vnegw	v3.0			361	Vector Negate Word VX-form
000100 101111	I	..XX			vnmsubfp	v2.03			412	Vector Negative Multiply-Subtract Floating-Point VA-form
000100 .... 10100 000100	I	..XX			vnor	v2.03			394	Vector Logical NOR VX-form
000100 .... 10010 000100	I	..XX			vor	v2.03			393	Vector Logical OR VX-form
000100 .... 10101 000100	I	..XX			vorc	v2.07			393	Vector Logical OR with Complement VX-form
000100 .... 10111 001101	I	..XX			vpdepd	v3.1			442	Vector Parallel Bits Deposit Doubleword VX-form
000100 101011	I	..XX			vperm	v2.03			286	Vector Permute VA-form
000100 111011	I	..XX			vpermr	v3.0			286	Vector Permute Right-indexed VA-form
000100 101101	I	..XX			vpermxor	v2.07			432	Vector Permute & Exclusive-OR VA-form
000100 .... 10110 001101	I	..XX			vpextd	v3.1			443	Vector Parallel Bits Extract Doubleword VX-form
000100 .... 01100 001110	I	..XX			vpkpx	v2.03			268	Vector Pack Pixel VX-form
000100 .... 10111 001110	I	..XX			vpksdss	v2.07			271	Vector Pack Signed Doubleword Signed Saturate VX-form
000100 .... 10101 001110	I	..XX			vpksdus	v2.07			271	Vector Pack Signed Doubleword Unsigned Saturate VX-form
000100 .... 00110 001110	I	..XX			vpkshss	v2.03			269	Vector Pack Signed Halfword Signed Saturate VX-form
000100 .... 00100 001110	I	..XX			vpkshus	v2.03			269	Vector Pack Signed Halfword Unsigned Saturate VX-form
000100 .... 00111 001110	I	..XX			vpkswss	v2.03			270	Vector Pack Signed Word Signed Saturate VX-form
000100 .... 00101 001110	I	..XX			vpkswus	v2.03			270	Vector Pack Signed Word Unsigned Saturate VX-form
000100 .... 10001 001110	I	..XX			vpkudum	v2.07			274	Vector Pack Unsigned Doubleword Unsigned Modulo VX-form
000100 .... 10011 001110	I	..XX			vpkudus	v2.07			274	Vector Pack Unsigned Doubleword Unsigned Saturate VX-form
000100 .... 00000 001110	I	..XX			vpkuhum	v2.03			272	Vector Pack Unsigned Halfword Unsigned Modulo VX-form
000100 .... 00010 001110	I	..XX			vpkuhus	v2.03			272	Vector Pack Unsigned Halfword Unsigned Saturate VX-form
000100 .... 00001 001110	I	..XX			vpkuwum	v2.03			273	Vector Pack Unsigned Word Unsigned Modulo VX-form
000100 .... 00011 001110	I	..XX			vpkuwus	v2.03			273	Vector Pack Unsigned Word Unsigned Saturate VX-form
000100 .... 10000 001000	I	..XX			vpmsumb	v2.07			430	Vector Polynomial Multiply-Sum Byte VX-form
000100 .... 10011 001000	I	..XX			vpmsumd	v2.07			431	Vector Polynomial Multiply-Sum Doubleword VX-form
000100 .... 10001 001000	I	..XX			vpmsumh	v2.07			430	Vector Polynomial Multiply-Sum Halfword VX-form
000100 .... 10010 001000	I	..XX			vpmsumw	v2.07			431	Vector Polynomial Multiply-Sum Word VX-form
000100 ..... ///// 11100 000011	I	..XX			vpopcntb	v2.07			445	Vector Population Count Byte VX-form
000100 ..... ///// 11111 000011	I	..XX			vpopcntd	v2.07			446	Vector Population Count Doubleword VX-form
000100 ..... ///// 11101 000011	I	..XX			vpopcnth	v2.07			445	Vector Population Count Halfword VX-form
000100 ..... ///// 11110 000011	I	..XX			vpopcntw	v2.07			446	Vector Population Count Word VX-form
000100 ..... 01001 ..... 11000 000010	I	..XX			vprtybd	v3.0			447	Vector Parity Byte Doubleword VX-form
000100 ..... 01010 ..... 11000 000010	I	..XX			vprtybq	v3.0			448	Vector Parity Byte Quadword VX-form
000100 ..... 01000 ..... 11000 000010	I	..XX			vprtybw	v3.0			447	Vector Parity Byte Word VX-form
000100 ..... ///// 00100 001010	I	..XX			vrefp	v2.03			423	Vector Reciprocal Estimate Floating-Point VX-form
000100 ..... ///// 01011 001010	I	..XX			vrfim	v2.03			416	Vector Round to Floating-Point Integer toward -Infinity VX-form
000100 ..... ///// 01000 001010	I	..XX			vrfin	v2.03			416	Vector Round to Floating-Point Integer Nearest VX-form
000100 ..... ///// 01010 001010	I	..XX			vrfip	v2.03			417	Vector Round to Floating-Point Integer toward +Infinity VX-form
000100 ..... ///// 01001 001010	I	..XX			vrfiz	v2.03			417	Vector Round to Floating-Point Integer toward Zero VX-form
000100 .... 00000 000100	I	..XX			vrlb	v2.03			395	Vector Rotate Left Byte VX-form
000100 .... 00011 000100	I	..XX			vrld	v2.07			396	Vector Rotate Left Doubleword VX-form

Table H.1: Power ISA Instruction Set Sorted by Mnemonic (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
000100 .... 00011 000101	I	..XX			vrldmi	v3.0			401	Vector Rotate Left Doubleword then Mask Insert VX-form
000100 .... 00111 000101	I	..XX			vrldnm	v3.0			398	Vector Rotate Left Doubleword then AND with Mask VX-form
000100 .... 00001 000100	I	..XX			vrlh	v2.03			395	Vector Rotate Left Halfword VX-form
000100 .... 00000 000101	I	..XX			vrlq	v3.1			397	Vector Rotate Left Quadword VX-form
000100 .... 00001 000101	I	..XX			vrlqmi	v3.1			401	Vector Rotate Left Quadword then Mask Insert VX-form
000100 .... 00101 000101	I	..XX			vrlqnm	v3.1			399	Vector Rotate Left Quadword then AND with Mask VX-form
000100 .... 00010 000100	I	..XX			vrlw	v2.03			396	Vector Rotate Left Word VX-form
000100 .... 00010 000101	I	..XX			vrlwmi	v3.0			400	Vector Rotate Left Word then Mask Insert VX-form
000100 .... 00110 000101	I	..XX			vrlwnm	v3.0			398	Vector Rotate Left Word then AND with Mask VX-form
000100 ..... ///// 00101 001010	I	..XX			vrsqrtefp	v2.03			423	Vector Reciprocal Square Root Estimate Floating-Point VX-form
000100 .... ///// 10111 001000	I	..XX			vsbox	v2.07			426	Vector AES SubBytes VX-form
000100 101010	I	..XX			vsel	v2.03			287	Vector Select VA-form
000100 .... 11011 000010	I	..XX			vshasigmad	v2.07			427	Vector SHA-512 Sigma Doubleword VX-form
000100 .... 11010 000010	I	..XX			vshasigmaw	v2.07			428	Vector SHA-256 Sigma Word VX-form
000100 .... 00111 000100	I	..XX			vsl	v2.03			290	Vector Shift Left VX-form
000100 .... 00100 000100	I	..XX			vslb	v2.03			402	Vector Shift Left Byte VX-form
000100 .... 10111 000100	I	..XX			vsld	v2.07			403	Vector Shift Left Doubleword VX-form
000100 .... 00... 010110	I	..XX			vsldbi	v3.1			289	Vector Shift Left Double by Bit Immediate VN-form
000100 .... /.... 101100	I	..XX			vsldoi	v2.03			289	Vector Shift Left Double by Octet Immediate VA-form
000100 .... 00101 000100	I	..XX			vslh	v2.03			402	Vector Shift Left Halfword VX-form
000100 .... 10000 001100	I	..XX			vslo	v2.03			291	Vector Shift Left by Octet VX-form
000100 .... 00100 000101	I	..XX			vslq	v3.1			404	Vector Shift Left Quadword VX-form
000100 .... 11101 000100	I	..XX			vslv	v3.0			292	Vector Shift Left Variable VX-form
000100 .... 00110 000100	I	..XX			vslw	v2.03			403	Vector Shift Left Word VX-form
000100 ..... /.... ..... 01000 001100	I	..XX			vspltb	v2.03			283	Vector Splat Byte VX-form
000100 ..... //... ..... 01001 001100	I	..XX			vsplth	v2.03			283	Vector Splat Halfword VX-form
000100 .... ///// 01100 001100	I	..XX			vspltisb	v2.03			285	Vector Splat Immediate Signed Byte VX-form
000100 .... ///// 01101 001100	I	..XX			vspltish	v2.03			285	Vector Splat Immediate Signed Halfword VX-form
000100 .... ///// 01110 001100	I	..XX			vspltisw	v2.03			285	Vector Splat Immediate Signed Word VX-form
000100 ..... ///.. ..... 01010 001100	I	..XX			vspltw	v2.03			284	Vector Splat Word VX-form
000100 .... 01011 000100	I	..XX			vsr	v2.03			290	Vector Shift Right VX-form
000100 .... 01100 000100	I	..XX			vsrab	v2.03			408	Vector Shift Right Algebraic Byte VX-form
000100 .... 01111 000100	I	..XX			vsrad	v2.07			409	Vector Shift Right Algebraic Doubleword VX-form
000100 .... 01101 000100	I	..XX			vsrah	v2.03			408	Vector Shift Right Algebraic Halfword VX-form
000100 .... 01100 000101	I	..XX			vsraq	v3.1			410	Vector Shift Right Algebraic Quadword VX-form
000100 .... 01110 000100	I	..XX			vsraw	v2.03			409	Vector Shift Right Algebraic Word VX-form
000100 .... 01000 000100	I	..XX			vsrb	v2.03			405	Vector Shift Right Byte VX-form
000100 .... 11011 000100	I	..XX			vsrd	v2.07			406	Vector Shift Right Doubleword VX-form
000100 .... 01... 010110	I	..XX			vsrdbi	v3.1			289	Vector Shift Right Double by Bit Immediate VN-form
000100 .... 01001 000100	I	..XX			vsrh	v2.03			405	Vector Shift Right Halfword VX-form
000100 .... 10001 001100	I	..XX			vsro	v2.03			291	Vector Shift Right by Octet VX-form
000100 .... 01000 000101	I	..XX			vsrq	v3.1			407	Vector Shift Right Quadword VX-form
000100 .... 11100 000100	I	..XX			vsrv	v3.0			292	Vector Shift Right Variable VX-form
000100 .... 01010 000100	I	..XX			vsrw	v2.03			406	Vector Shift Right Word VX-form
000100 ..... 00000 ..... .0000 001101	I	..XX			vstribl[.]	v3.1			462	Vector String Isolate Byte Left-justified VC-form

Table H.1: Power ISA Instruction Set Sorted by Mnemonic (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
000100 ..... 00001 ..... .0000 001101	I	..XX			vstribr[.]	v3.1			462	Vector String Isolate Byte Right-justified VC-form
000100 ..... 00010 ..... .0000 001101	I	..XX			vstrihl[.]	v3.1			463	Vector String Isolate Halfword Left-justified VC-form
000100 ..... 00011 ..... .0000 001101	I	..XX			vstrihr[.]	v3.1			463	Vector String Isolate Halfword Right-justified VC-form
000100 .... 10101 000000	I	..XX			vsubcuq	v2.07			328	Vector Subtract & write Carry-out Unsigned Quadword VX-form
000100 .... 10110 000000	I	..XX			vsubcuw	v2.03			321	Vector Subtract & Write Carry-out Unsigned Word VX-form
000100 111111	I	..XX			vsubecuq	v2.07			328	Vector Subtract Extended & write Carry-out Unsigned Quadword VA-form
000100 111110	I	..XX			vsubeuqm	v2.07			327	Vector Subtract Extended Unsigned Quadword Modulo VA-form
000100 .... 00001 001010	I	..XX			vsubfp	v2.03			411	Vector Subtract Floating-Point VX-form
000100 .... 11100 000000	I	..XX			vsubsbs	v2.03			321	Vector Subtract Signed Byte Saturate VX-form
000100 .... 11101 000000	I	..XX			vsubshs	v2.03			322	Vector Subtract Signed Halfword Saturate VX-form
000100 .... 11110 000000	I	..XX			vsubsws	v2.03			322	Vector Subtract Signed Word Saturate VX-form
000100 .... 10000 000000	I	..XX			vsububm	v2.03			323	Vector Subtract Unsigned Byte Modulo VX-form
000100 .... 11000 000000	I	..XX			vsububs	v2.03			325	Vector Subtract Unsigned Byte Saturate VX-form
000100 .... 10011 000000	I	..XX			vsubudm	v2.07			324	Vector Subtract Unsigned Doubleword Modulo VX-form
000100 .... 10001 000000	I	..XX			vsubuhm	v2.03			323	Vector Subtract Unsigned Halfword Modulo VX-form
000100 .... 11001 000000	I	..XX			vsubuhs	v2.03			325	Vector Subtract Unsigned Halfword Saturate VX-form
000100 .... 10100 000000	I	..XX			vsubuqm	v2.07			327	Vector Subtract Unsigned Quadword Modulo VX-form
000100 .... 10010 000000	I	..XX			vsubuwm	v2.03			324	Vector Subtract Unsigned Word Modulo VX-form
000100 .... 11010 000000	I	..XX			vsubuws	v2.03			326	Vector Subtract Unsigned Word Saturate VX-form
000100 .... 11010 001000	I	..XX			vsum2sws	v2.03			358	Vector Sum across Half Signed Word Saturate VX-form
000100 .... 11100 001000	I	..XX			vsum4sbs	v2.03			359	Vector Sum across Quarter Signed Byte Saturate VX-form
000100 .... 11001 001000	I	..XX			vsum4shs	v2.03			359	Vector Sum across Quarter Signed Halfword Saturate VX-form
000100 .... 11000 001000	I	..XX			vsum4ubs	v2.03			360	Vector Sum across Quarter Unsigned Byte Saturate VX-form
000100 .... 11110 001000	I	..XX			vsumsws	v2.03			357	Vector Sum across Signed Word Saturate VX-form
000100 ..... ///// 01101 001110	I	..XX			vupkhpx	v2.03			278	Vector Unpack High Pixel VX-form
000100 ..... ///// 01000 001110	I	..XX			vupkhsb	v2.03			275	Vector Unpack High Signed Byte VX-form
000100 ..... ///// 01001 001110	I	..XX			vupkhsh	v2.03			276	Vector Unpack High Signed Halfword VX-form
000100 ..... ///// 11001 001110	I	..XX			vupkhsw	v2.07			277	Vector Unpack High Signed Word VX-form
000100 ..... ///// 01111 001110	I	..XX			vupklpx	v2.03			278	Vector Unpack Low Pixel VX-form
000100 ..... ///// 01010 001110	I	..XX			vupklsb	v2.03			275	Vector Unpack Low Signed Byte VX-form
000100 ..... ///// 01011 001110	I	..XX			vupklsh	v2.03			276	Vector Unpack Low Signed Halfword VX-form
000100 ..... ///// 11011 001110	I	..XX			vupklsw	v2.07			277	Vector Unpack Low Signed Word VX-form
000100 .... 10011 000100	I	..XX			vxor	v2.03			394	Vector Logical XOR VX-form
011111 ///.. ///.. ///// 00000 11110/	II	..XX			wait	v2.03			1029	Wait X-form
011111 .... 01001 11100.	I	XXXX			xor[.]	P1		SR	99	XOR X-form
011010	I	XXXX			xori	P1			98	XOR Immediate D-form
011011	I	XXXX			xoris	P1			98	XOR Immediate Shifted D-form
111100 ..... ///// ..... 10101 1001..	I	..XX			xsabsdp	v2.06			607	VSX Scalar Absolute Double-Precision XX2-form
111111 ..... 00000 ..... 11001 00100/	I	...X	BFP128		xsabsqp	v3.0			607	VSX Scalar Absolute Quad-Precision X-form
111100 .... 00100 000...	I	..XX			xsadddp	v2.06			615	VSX Scalar Add Double-Precision XX3-form
111111 .... 00000 00100.	I	...X	BFP128		xsaddqp[o]	v3.0			620	VSX Scalar Add Quad-Precision [using round to Odd] X-form

Table H.1: Power ISA Instruction Set Sorted by Mnemonic (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
111100 .... 00000 000...	I	..XX			xsaddsp	v2.07			622	VSX Scalar Add Single-Precision XX3-form
111100 .... 00000 011...	I	..XX			xscmpeqdp	v3.0			749	VSX Scalar Compare Equal Double-Precision XX3-form
111111 .... 00010 00100/	I	..XX			xscmpeqqp	v3.1			750	VSX Scalar Compare Equal Quad-Precision X-form
111100 ...// .... 00111 011../	I	..XX			xscmpexpdp	v3.0			862	VSX Scalar Compare Exponents Double-Precision XX3-form
111111 ...// 00101 00100/	I	...X	BFP128		xscmpexpqp	v3.0			863	VSX Scalar Compare Exponents Quad-Precision X-form
111100 .... 00010 011...	I	..XX			xscmpgedp	v3.0			751	VSX Scalar Compare Greater Than or Equal Double-Precision XX3-form
111111 .... 00110 00100/	I	..XX			xscmpgeqp	v3.1			752	VSX Scalar Compare Greater Than or Equal Quad-Precision X-form
111100 .... 00001 011...	I	..XX			xscmpgtdp	v3.0			753	VSX Scalar Compare Greater Than Double-Precision XX3-form
111111 .... 00111 00100/	I	..XX			xscmpgtqp	v3.1			754	VSX Scalar Compare Greater Than Quad-Precision X-form
111100 ...// .... 00101 011../	I	..XX			xscmpodp	v2.06			743	VSX Scalar Compare Ordered Double-Precision XX3-form
111111 ...// 00100 00100/	I	...X	BFP128		xscmpoqp	v3.0			745	VSX Scalar Compare Ordered Quad-Precision X-form
111100 ...// .... 00100 011../	I	..XX			xscmpudp	v2.06			746	VSX Scalar Compare Unordered Double-Precision XX3-form
111111 ...// 10100 00100/	I	...X	BFP128		xscmpuqp	v3.0			748	VSX Scalar Compare Unordered Quad-Precision X-form
111100 .... 10110 000...	I	..XX			xscpsgndp	v2.06			608	VSX Scalar Copy Sign Double-Precision XX3-form
111111 .... 00011 00100/	I	...X	BFP128		xscpsgnqp	v3.0			608	VSX Scalar Copy Sign Quad-Precision X-form
111100 ..... 10001 ..... 10101 1011..	I	..XX			xscvdphp	v3.0			788	VSX Scalar Convert with round Double-Precision to Half-Precision format XX2-form
111111 ..... 10110 ..... 11010 00100/	I	...X	BFP128		xscvdpqp	v3.0			795	VSX Scalar Convert Double-Precision to Quad-Precision format X-form
111100 ..... ///// ..... 10000 1001..	I	..XX			xscvdpsp	v2.06			789	VSX Scalar Convert with round Double-Precision to Single-Precision format XX2-form
111100 ..... ///// ..... 10000 1011..	I	..XX			xscvdpspn	v2.07			790	VSX Scalar Convert Scalar Single-Precision to Vector Single-Precision format Non-signalling XX2-form
111100 ..... ///// ..... 10101 1000..	I	..XX			xscvdpsxds	v2.06			816	VSX Scalar Convert with round to zero Double-Precision to Signed Doubleword format XX2-form
111100 ..... ///// ..... 00101 1000..	I	..XX			xscvdpsxws	v2.06			818	VSX Scalar Convert with round to zero Double-Precision to Signed Word format XX2-form
111100 ..... ///// ..... 10100 1000..	I	..XX			xscvdpuxds	v2.06			820	VSX Scalar Convert with round to zero Double-Precision to Unsigned Doubleword format XX2-form
111100 ..... ///// ..... 00100 1000..	I	..XX			xscvdpuxws	v2.06			822	VSX Scalar Convert with round to zero Double-Precision to Unsigned Word format XX2-form
111100 ..... 10000 ..... 10101 1011..	I	..XX			xscvhpdp	v3.0			796	VSX Scalar Convert Half-Precision to Double-Precision format XX2-form
111111 ..... 10100 ..... 11010 00100.	I	...X	BFP128		xscvqpdp[o]	v3.0			791	VSX Scalar Convert with round Quad-Precision to Double-Precision format [using round to Odd] X-form
111111 ..... 11001 ..... 11010 00100/	I	...X	BFP128		xscvqpsdz	v3.0			824	VSX Scalar Convert with round to zero Quad-Precision to Signed Doubleword format X-form
111111 ..... 01000 ..... 11010 00100/	I	..XX			xscvqpsqz	v3.1			826	VSX Scalar Convert with round to zero Quad-Precision to Signed Quadword X-form
111111 ..... 01001 ..... 11010 00100/	I	...X	BFP128		xscvqpswz	v3.0			828	VSX Scalar Convert with round to zero Quad-Precision to Signed Word format X-form
111111 ..... 10001 ..... 11010 00100/	I	...X	BFP128		xscvqpudz	v3.0			830	VSX Scalar Convert with round to zero Quad-Precision to Unsigned Doubleword format X-form

Table H.1: Power ISA Instruction Set Sorted by Mnemonic (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
111111 ..... 00000 ..... 11010 00100/	I	..XX			xscvqpuqz	v3.1			832	VSX Scalar Convert with round to zero Quad-Precision to Unsigned Quadword X-form
111111 ..... 00001 ..... 11010 00100/	I	...X	BFP128		xscvqpuwz	v3.0			834	VSX Scalar Convert with round to zero Quad-Precision to Unsigned Word format X-form
111111 ..... 01010 ..... 11010 00100/	I	...X	BFP128		xscvsdqp	v3.0			852	VSX Scalar Convert Signed Doubleword to Quad-Precision format X-form
111100 ..... ///// ..... 10100 1001..	I	..XX			xscvspdp	v2.06			797	VSX Scalar Convert Single-Precision to Double-Precision format XX2-form
111100 ..... ///// ..... 10100 1011..	I	..XX			xscvspdpn	v2.07			798	VSX Scalar Convert Single-Precision to Double-Precision format Non-signalling XX2-form
111111 ..... 01011 ..... 11010 00100/	I	..XX			xscvsqqp	v3.1			853	VSX Scalar Convert with round Signed Quadword to Quad-Precision X-form
111100 ..... ///// ..... 10111 1000..	I	..XX			xscvsxddp	v2.06			854	VSX Scalar Convert with round Signed Doubleword to Double-Precision format XX2-form
111100 ..... ///// ..... 10011 1000..	I	..XX			xscvsxdsp	v2.07			855	VSX Scalar Convert with round Signed Doubleword to Single-Precision format XX2-form
111111 ..... 00010 ..... 11010 00100/	I	...X	BFP128		xscvudqp	v3.0			852	VSX Scalar Convert Unsigned Doubleword to Quad-Precision format X-form
111111 ..... 00011 ..... 11010 00100/	I	..XX			xscvuqqp	v3.1			853	VSX Scalar Convert with round Unsigned Quadword to Quad-Precision format X-form
111100 ..... ///// ..... 10110 1000..	I	..XX			xscvuxddp	v2.06			854	VSX Scalar Convert with round Unsigned Doubleword to Double-Precision format XX2-form
111100 ..... ///// ..... 10010 1000..	I	..XX			xscvuxdsp	v2.07			855	VSX Scalar Convert with round Unsigned Doubleword to Single-Precision format XX2-form
111100 .... 00111 000...	I	..XX			xsdivdp	v2.06			624	VSX Scalar Divide Double-Precision XX3-form
111111 .... 10001 00100.	I	...X	BFP128		xsdivqp[o]	v3.0			626	VSX Scalar Divide Quad-Precision [using round to Odd] X-form
111100 .... 00011 000...	I	..XX			xsdivsp	v2.07			628	VSX Scalar Divide Single-Precision XX3-form
111100 .... 11100 10110.	I	..XX			xsiexpdp	v3.0			864	VSX Scalar Insert Exponent Double-Precision X-form
111111 .... 11011 00100/	I	...X	BFP128		xsiexpqp	v3.0			865	VSX Scalar Insert Exponent Quad-Precision X-form
111100 .... 00100 001...	I	..XX			xsmaddadp	v2.06			648	VSX Scalar Multiply-Add Type-A Double-Precision XX3-form
111100 .... 00000 001...	I	..XX			xsmaddasp	v2.07			651	VSX Scalar Multiply-Add Type-A Single-Precision XX3-form
111100 .... 00101 001...	I	..XX			xsmaddmdp	v2.06			648	VSX Scalar Multiply-Add Type-M Double-Precision XX3-form
111100 .... 00001 001...	I	..XX			xsmaddmsp	v2.07			651	VSX Scalar Multiply-Add Type-M Single-Precision XX3-form
111111 .... 01100 00100.	I	...X	BFP128		xsmaddqp[o]	v3.0			654	VSX Scalar Multiply-Add Quad-Precision [using round to Odd] X-form
111100 .... 10000 000...	I	..XX			xsmaxcdp	v3.0			755	VSX Scalar Maximum Type-C Double-Precision XX3-form
111111 .... 10101 00100/	I	..XX			xsmaxcqp	v3.1			757	VSX Scalar Maximum Type-C Quad-Precision X-form
111100 .... 10100 000...	I	..XX			xsmaxdp	v2.06			759	VSX Scalar Maximum Double-Precision XX3-form
111100 .... 10010 000...	I	..XX			xsmaxjdp	v3.0			761	VSX Scalar Maximum Type-J Double-Precision XX3-form
111100 .... 10001 000...	I	..XX			xsmincdp	v3.0			763	VSX Scalar Minimum Type-C Double-Precision XX3-form
111111 .... 10111 00100/	I	..XX			xsmincqp	v3.1			765	VSX Scalar Minimum Type-C Quad-Precision X-form
111100 .... 10101 000...	I	..XX			xsmindp	v2.06			767	VSX Scalar Minimum Double-Precision XX3-form
111100 .... 10011 000...	I	..XX			xsminjdp	v3.0			769	VSX Scalar Minimum Type-J Double-Precision XX3-form

Table H.1: Power ISA Instruction Set Sorted by Mnemonic (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
111100 .... 00110 001...	I	..XX			xsmsubadp	v2.06			657	VSX Scalar Multiply-Subtract Type-A Double-Precision XX3-form
111100 .... 00010 001...	I	..XX			xsmsubasp	v2.07			660	VSX Scalar Multiply-Subtract Type-A Single-Precision XX3-form
111100 .... 00111 001...	I	..XX			xsmsubmdp	v2.06			657	VSX Scalar Multiply-Subtract Type-M Double-Precision XX3-form
111100 .... 00011 001...	I	..XX			xsmsubmsp	v2.07			660	VSX Scalar Multiply-Subtract Type-M Single-Precision XX3-form
111111 .... 01101 00100.	I	...X	BFP128		xsmsubqp[o]	v3.0			663	VSX Scalar Multiply-Subtract Quad-Precision [using round to Odd] X-form
111100 .... 00110 000...	I	..XX			xsmuldp	v2.06			630	VSX Scalar Multiply Double-Precision XX3-form
111111 .... 00001 00100.	I	...X	BFP128		xsmulqp[o]	v3.0			632	VSX Scalar Multiply Quad-Precision [using round to Odd] X-form
111100 .... 00010 000...	I	..XX			xsmulsp	v2.07			634	VSX Scalar Multiply Single-Precision XX3-form
111100 ..... ///// ..... 10110 1001..	I	..XX			xsnabsdp	v2.06			609	VSX Scalar Negative Absolute Double-Precision XX2-form
111111 ..... 01000 ..... 11001 00100/	I	...X	BFP128		xsnabsqp	v3.0			609	VSX Scalar Negative Absolute Quad-Precision X-form
111100 ..... ///// ..... 10111 1001..	I	..XX			xsnegdp	v2.06			610	VSX Scalar Negate Double-Precision XX2-form
111111 ..... 10000 ..... 11001 00100/	I	...X	BFP128		xsnegqp	v3.0			610	VSX Scalar Negate Quad-Precision X-form
111100 .... 10100 001...	I	..XX			xsnmaddadp	v2.06			666	VSX Scalar Negative Multiply-Add Type-A Double-Precision XX3-form
111100 .... 10000 001...	I	..XX			xsnmaddasp	v2.07			670	VSX Scalar Negative Multiply-Add Type-A Single-Precision XX3-form
111100 .... 10101 001...	I	..XX			xsnmaddmdp	v2.06			666	VSX Scalar Negative Multiply-Add Type-M Double-Precision XX3-form
111100 .... 10001 001...	I	..XX			xsnmaddmsp	v2.07			670	VSX Scalar Negative Multiply-Add Type-M Single-Precision XX3-form
111111 .... 01110 00100.	I	...X	BFP128		xsnmaddqp[o]	v3.0			673	VSX Scalar Negative Multiply-Add Quad-Precision [using round to Odd] X-form
111100 .... 10110 001...	I	..XX			xsnmsubadp	v2.06			676	VSX Scalar Negative Multiply-Subtract Type-A Double-Precision XX3-form
111100 .... 10010 001...	I	..XX			xsnmsubasp	v2.07			679	VSX Scalar Negative Multiply-Subtract Type-A Single-Precision XX3-form
111100 .... 10111 001...	I	..XX			xsnmsubmdp	v2.06			676	VSX Scalar Negative Multiply-Subtract Type-M Double-Precision XX3-form
111100 .... 10011 001...	I	..XX			xsnmsubmsp	v2.07			679	VSX Scalar Negative Multiply-Subtract Type-M Single-Precision XX3-form
111111 .... 01111 00100.	I	...X	BFP128		xsnmsubqp[o]	v3.0			682	VSX Scalar Negative Multiply-Subtract Quad-Precision [using round to Odd] X-form
111100 ..... ///// ..... 00100 1001..	I	..XX			xsrdpi	v2.06			801	VSX Scalar Round to Double-Precision Integer using round to Nearest Away XX2-form
111100 ..... ///// ..... 00110 1011..	I	..XX			xsrdpic	v2.06			802	VSX Scalar Round to Double-Precision Integer exact using Current rounding mode XX2-form
111100 ..... ///// ..... 00111 1001..	I	..XX			xsrdpim	v2.06			803	VSX Scalar Round to Double-Precision Integer using round toward -Infinity XX2-form
111100 ..... ///// ..... 00110 1001..	I	..XX			xsrdpip	v2.06			804	VSX Scalar Round to Double-Precision Integer using round toward +Infinity XX2-form
111100 ..... ///// ..... 00101 1001..	I	..XX			xsrdpiz	v2.06			805	VSX Scalar Round to Double-Precision Integer using round toward Zero XX2-form

Table H.1: Power ISA Instruction Set Sorted by Mnemonic (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
111100 ..... ///// ..... 00101 1010..	I	..XX			xsredp	v2.06			685	VSX Scalar Reciprocal Estimate Double-Precision XX2-form
111100 ..... ///// ..... 00001 1010..	I	..XX			xsresp	v2.07			686	VSX Scalar Reciprocal Estimate Single-Precision XX2-form
111111 ..... ////. ....000 00101.	I	...X	BFP128		xsrqpi[x]	v3.0			806	VSX Scalar Round to Quad-Precision Integer [with Inexact] Z23-form
111111 ..... ////. 001 00101/	I	...X	BFP128		xsrqpxp	v3.0			785	VSX Scalar Round Quad-Precision to Double-Extended-Precision Z23-form
111100 ..... ///// ..... 10001 1001..	I	..XX			xsrsp	v2.07			787	VSX Scalar Round to Single-Precision XX2-form
111100 ..... ///// ..... 00100 1010..	I	..XX			xsrsqrtedp	v2.06			687	VSX Scalar Reciprocal Square Root Estimate Double-Precision XX2-form
111100 ..... ///// ..... 00000 1010..	I	..XX			xsrsqrtesp	v2.07			688	VSX Scalar Reciprocal Square Root Estimate Single-Precision XX2-form
111100 ..... ///// ..... 00100 1011..	I	..XX			xssqrtdp	v2.06			636	VSX Scalar Square Root Double-Precision XX2-form
111111 ..... 11011 ..... 11001 00100.	I	...X	BFP128		xssqrtqp[o]	v3.0			638	VSX Scalar Square Root Quad-Precision [using round to Odd] X-form
111100 ..... ///// ..... 00000 1011..	I	..XX			xssqrtsp	v2.07			640	VSX Scalar Square Root Single-Precision XX2-form
111100 .... 00101 000...	I	..XX			xssubdp	v2.06			642	VSX Scalar Subtract Double-Precision XX3-form
111111 .... 10000 00100.	I	...X	BFP128		xssubqp[o]	v3.0			644	VSX Scalar Subtract Quad-Precision [using round to Odd] X-form
111100 .... 00001 000...	I	..XX			xssubsp	v2.07			646	VSX Scalar Subtract Single-Precision XX3-form
111100 ...// .... 00111 101../	I	..XX			xstdivdp	v2.06			689	VSX Scalar Test for software Divide Double-Precision XX3-form
111100 ...// ///// 00110 1010./	I	..XX			xstsqrtdp	v2.06			690	VSX Scalar Test for software Square Root Double-Precision XX2-form
111100 .... 10110 1010./	I	..XX			xststdcdp	v3.0			866	VSX Scalar Test Data Class Double-Precision XX2-form
111111 .... 10110 00100/	I	...X	BFP128		xststdcqp	v3.0			867	VSX Scalar Test Data Class Quad-Precision X-form
111100 .... 10010 1010./	I	..XX			xststdcsp	v3.0			868	VSX Scalar Test Data Class Single-Precision XX2-form
111100 ..... 00000 ..... 10101 1011./	I	..XX			xsxexpdp	v3.0			869	VSX Scalar Extract Exponent Double-Precision XX2-form
111111 ..... 00010 ..... 11001 00100/	I	...X	BFP128		xsxexpqp	v3.0			869	VSX Scalar Extract Exponent Quad-Precision X-form
111100 ..... 00001 ..... 10101 1011./	I	..XX			xsxsigdp	v3.0			870	VSX Scalar Extract Significand Double-Precision XX2-form
111111 ..... 10010 ..... 11001 00100/	I	...X	BFP128		xsxsigqp	v3.0			870	VSX Scalar Extract Significand Quad-Precision X-form
111100 ..... ///// ..... 11101 1001..	I	..XX			xvabsdp	v2.06			611	VSX Vector Absolute Double-Precision XX2-form
111100 ..... ///// ..... 11001 1001..	I	..XX			xvabssp	v2.06			611	VSX Vector Absolute Single-Precision XX2-form
111100 .... 01100 000...	I	..XX			xvadddp	v2.06			691	VSX Vector Add Double-Precision XX3-form
111100 .... 01000 000...	I	..XX			xvaddsp	v2.06			694	VSX Vector Add Single-Precision XX3-form
111011 ...// .... 00110 011../	I	....	MMA	MMA	xvbf16ger2	v3.1			889	VSX Vector bfloat16 GER (rank-2 update) XX3-form
111011 ...// .... 11110 010../	I	....	MMA	MMA	xvbf16ger2nn	v3.1			889	VSX Vector bfloat16 GER (rank-2 update) Negative multiply, Negative accumulate XX3-form
111011 ...// .... 01110 010../	I	....	MMA	MMA	xvbf16ger2np	v3.1			889	VSX Vector bfloat16 GER (rank-2 update) Negative multiply, Positive accumulate XX3-form
111011 ...// .... 10110 010../	I	....	MMA	MMA	xvbf16ger2pn	v3.1			889	VSX Vector bfloat16 GER (rank-2 update) Positive multiply, Negative accumulate XX3-form
111011 ...// .... 00110 010../	I	....	MMA	MMA	xvbf16ger2pp	v3.1			889	VSX Vector bfloat16 GER (rank-2 update) Positive multiply, Positive accumulate XX3-form
111100 .... .1100 011...	I	..XX			xvcmpeqdp[.]	v2.06			771	VSX Vector Compare Equal To Double-Precision XX3-form
111100 .... .1000 011...	I	..XX			xvcmpeqsp[.]	v2.06			772	VSX Vector Compare Equal To Single-Precision XX3-form

Table H.1: Power ISA Instruction Set Sorted by Mnemonic (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
111100 .... .1110 011...	I	..XX			xvcmpgedp[.]	v2.06			773	VSX Vector Compare Greater Than or Equal To Double-Precision XX3-form
111100 .... .1010 011...	I	..XX			xvcmpgesp[.]	v2.06			774	VSX Vector Compare Greater Than or Equal To Single-Precision XX3-form
111100 .... .1101 011...	I	..XX			xvcmpgtdp[.]	v2.06			775	VSX Vector Compare Greater Than Double-Precision XX3-form
111100 .... .1001 011...	I	..XX			xvcmpgtsp[.]	v2.06			776	VSX Vector Compare Greater Than Single-Precision XX3-form
111100 .... 11110 000...	I	..XX			xvcpsgndp	v2.06			612	VSX Vector Copy Sign Double-Precision XX3-form
111100 .... 11010 000...	I	..XX			xvcpsgnsp	v2.06			612	VSX Vector Copy Sign Single-Precision XX3-form
111100 ..... 10000 ..... 11101 1011..	I	..XX			xvcvbf16spn	v3.1			800	VSX Vector Convert bfloat16 to Single-Precision format Non-signaling XX2-form
111100 ..... ///// ..... 11000 1001..	I	..XX			xvcvdpsp	v2.06			793	VSX Vector Convert with round Double-Precision to Single-Precision format XX2-form
111100 ..... ///// ..... 11101 1000..	I	..XX			xvcvdpsxds	v2.06			836	VSX Vector Convert with round to zero Double-Precision to Signed Doubleword format XX2-form
111100 ..... ///// ..... 01101 1000..	I	..XX			xvcvdpsxws	v2.06			838	VSX Vector Convert with round to zero Double-Precision to Signed Word format XX2-form
111100 ..... ///// ..... 11100 1000..	I	..XX			xvcvdpuxds	v2.06			840	VSX Vector Convert with round to zero Double-Precision to Unsigned Doubleword format XX2-form
111100 ..... ///// ..... 01100 1000..	I	..XX			xvcvdpuxws	v2.06			842	VSX Vector Convert with round to zero Double-Precision to Unsigned Word format XX2-form
111100 ..... 11000 ..... 11101 1011..	I	..XX			xvcvhpsp	v3.0			799	VSX Vector Convert Half-Precision to Single-Precision format XX2-form
111100 ..... 10001 ..... 11101 1011..	I	..XX			xvcvspbf16	v3.1			792	VSX Vector Convert with round Single-Precision to bfloat16 format XX2-form
111100 ..... ///// ..... 11100 1001..	I	..XX			xvcvspdp	v2.06			800	VSX Vector Convert Single-Precision to Double-Precision format XX2-form
111100 ..... 11001 ..... 11101 1011..	I	..XX			xvcvsphp	v3.0			794	VSX Vector Convert with round Single-Precision to Half-Precision format XX2-form
111100 ..... ///// ..... 11001 1000..	I	..XX			xvcvspsxds	v2.06			844	VSX Vector Convert with round to zero Single-Precision to Signed Doubleword format XX2-form
111100 ..... ///// ..... 01001 1000..	I	..XX			xvcvspsxws	v2.06			846	VSX Vector Convert with round to zero Single-Precision to Signed Word format XX2-form
111100 ..... ///// ..... 11000 1000..	I	..XX			xvcvspuxds	v2.06			848	VSX Vector Convert with round to zero Single-Precision to Unsigned Doubleword format XX2-form
111100 ..... ///// ..... 01000 1000..	I	..XX			xvcvspuxws	v2.06			850	VSX Vector Convert with round to zero Single-Precision to Unsigned Word format XX2-form
111100 ..... ///// ..... 11111 1000..	I	..XX			xvcvsxddp	v2.06			857	VSX Vector Convert with round Signed Doubleword to Double-Precision format XX2-form
111100 ..... ///// ..... 11011 1000..	I	..XX			xvcvsxdsp	v2.06			859	VSX Vector Convert with round Signed Doubleword to Single-Precision format XX2-form
111100 ..... ///// ..... 01111 1000..	I	..XX			xvcvsxwdp	v2.06			858	VSX Vector Convert Signed Word to Double-Precision format XX2-form
111100 ..... ///// ..... 01011 1000..	I	..XX			xvcvsxwsp	v2.06			861	VSX Vector Convert with round Signed Word to Single-Precision format XX2-form
111100 ..... ///// ..... 11110 1000..	I	..XX			xvcvuxddp	v2.06			857	VSX Vector Convert with round Unsigned Doubleword to Double-Precision format XX2-form

Table H.1: Power ISA Instruction Set Sorted by Mnemonic (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
111100 ..... ///// ..... 11010 1000..	I	..XX			xvcvuxdsp	v2.06			859	VSX Vector Convert with round Unsigned Doubleword to Single-Precision format XX2-form
111100 ..... ///// ..... 01110 1000..	I	..XX			xvcvuxwdp	v2.06			858	VSX Vector Convert Unsigned Word to Double-Precision format XX2-form
111100 ..... ///// ..... 01010 1000..	I	..XX			xvcvuxwsp	v2.06			861	VSX Vector Convert with round Unsigned Word to Single-Precision format XX2-form
111100 .... 01111 000...	I	..XX			xvdivdp	v2.06			696	VSX Vector Divide Double-Precision XX3-form
111100 .... 01011 000...	I	..XX			xvdivsp	v2.06			698	VSX Vector Divide Single-Precision XX3-form
111011 .../// .... 00010 011../	I	....	MMA	MMA	xvf16ger2	v3.1			894	VSX Vector 16-bit Floating-Point GER (rank-2 update) XX3-form
111011 .../// .... 11010 010../	I	....	MMA	MMA	xvf16ger2nn	v3.1			894	VSX Vector 16-bit Floating-Point GER (rank-2 update) Negative multiply, Negative accumulate XX3-form
111011 .../// .... 01010 010../	I	....	MMA	MMA	xvf16ger2np	v3.1			894	VSX Vector 16-bit Floating-Point GER (rank-2 update) Negative multiply, Positive accumulate XX3-form
111011 .../// .... 10010 010../	I	....	MMA	MMA	xvf16ger2pn	v3.1			894	VSX Vector 16-bit Floating-Point GER (rank-2 update) Positive multiply, Negative accumulate XX3-form
111011 .../// .... 00010 010../	I	....	MMA	MMA	xvf16ger2pp	v3.1			894	VSX Vector 16-bit Floating-Point GER (rank-2 update) Positive multiply, Positive accumulate XX3-form
111011 .../// .... 00011 011../	I	....	MMA	MMA	xvf32ger	v3.1			899	VSX Vector 32-bit Floating-Point GER (rank-1 update) XX3-form
111011 .../// .... 11011 010../	I	....	MMA	MMA	xvf32gernn	v3.1			899	VSX Vector 32-bit Floating-Point GER (rank-1 update) Negative multiply, Negative accumulate XX3-form
111011 .../// .... 01011 010../	I	....	MMA	MMA	xvf32gernp	v3.1			899	VSX Vector 32-bit Floating-Point GER (rank-1 update) Negative multiply, Positive accumulate XX3-form
111011 .../// .... 10011 010../	I	....	MMA	MMA	xvf32gerpn	v3.1			899	VSX Vector 32-bit Floating-Point GER (rank-1 update) Positive multiply, Negative accumulate XX3-form
111011 .../// .... 00011 010../	I	....	MMA	MMA	xvf32gerpp	v3.1			899	VSX Vector 32-bit Floating-Point GER (rank-1 update) Positive multiply, Positive accumulate XX3-form
111011 .../// .... 00111 011../	I	....	MMA	MMA	xvf64ger	v3.1			903	VSX Vector 64-bit Floating-Point GER (rank-1 update) XX3-form
111011 .../// .... 11111 010../	I	....	MMA	MMA	xvf64gernn	v3.1			903	VSX Vector 64-bit Floating-Point GER (rank-1 update) Negative multiply, Negative accumulate XX3-form
111011 .../// .... 01111 010../	I	....	MMA	MMA	xvf64gernp	v3.1			903	VSX Vector 64-bit Floating-Point GER (rank-1 update) Negative multiply, Positive accumulate XX3-form
111011 .../// .... 10111 010../	I	....	MMA	MMA	xvf64gerpn	v3.1			903	VSX Vector 64-bit Floating-Point GER (rank-1 update) Positive multiply, Negative accumulate XX3-form
111011 .../// .... 00111 010../	I	....	MMA	MMA	xvf64gerpp	v3.1			903	VSX Vector 64-bit Floating-Point GER (rank-1 update) Positive multiply, Positive accumulate XX3-form
111011 .../// .... 01001 011../	I	....	MMA	MMA	xvi16ger2	v3.1			878	VSX Vector 16-bit Signed Integer GER (rank-2 update) XX3-form
111011 .../// .... 01101 011../	I	....	MMA	MMA	xvi16ger2pp	v3.1			878	VSX Vector 16-bit Signed Integer GER (rank-2 update) Positive multiply, Positive accumulate XX3-form
111011 .../// .... 00101 011../	I	....	MMA	MMA	xvi16ger2s	v3.1			880	VSX Vector 16-bit Signed Integer GER (rank-2 update) with Saturation XX3-form
111011 .../// .... 00101 010../	I	....	MMA	MMA	xvi16ger2spp	v3.1			880	VSX Vector 16-bit Signed Integer GER (rank-2 update) with Saturation Positive multiply, Positive accumulate XX3-form
111011 .../// .... 00100 011../	I	....	MMA	MMA	xvi4ger8	v3.1			882	VSX Vector 4-bit Signed Integer GER (rank-8 update) XX3-form

Table H.1: Power ISA Instruction Set Sorted by Mnemonic (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
111011 ...// .... 00100 010../	I	....	MMA	MMA	xvi4ger8pp	v3.1			882	VSX Vector 4-bit Signed Integer GER (rank-8 update) Positive multiply, Positive accumulate XX3-form
111011 ...// .... 00000 011../	I	....	MMA	MMA	xvi8ger4	v3.1			885	VSX Vector 8-bit Signed/Unsigned Integer GER (rank-4 update) XX3-form
111011 ...// .... 00000 010../	I	....	MMA	MMA	xvi8ger4pp	v3.1			885	VSX Vector 8-bit Signed/Unsigned Integer GER (rank-4 update) Positive multiply, Positive accumulate XX3-form
111011 ...// .... 01100 011../	I	....	MMA	MMA	xvi8ger4spp	v3.1			887	VSX Vector 8-bit Signed/Unsigned Integer GER (rank-4 update) with Saturation Positive multiply, Positive accumulate XX3-form
111100 .... 11111 000...	I	..XX			xviexpdp	v3.0			871	VSX Vector Insert Exponent Double-Precision XX3-form
111100 .... 11011 000...	I	..XX			xviexpsp	v3.0			871	VSX Vector Insert Exponent Single-Precision XX3-form
111100 .... 01100 001...	I	..XX			xvmaddadp	v2.06			710	VSX Vector Multiply-Add Type-A Double-Precision XX3-form
111100 .... 01000 001...	I	..XX			xvmaddasp	v2.06			713	VSX Vector Multiply-Add Type-A Single-Precision XX3-form
111100 .... 01101 001...	I	..XX			xvmaddmdp	v2.06			710	VSX Vector Multiply-Add Type-M Double-Precision XX3-form
111100 .... 01001 001...	I	..XX			xvmaddmsp	v2.06			713	VSX Vector Multiply-Add Type-M Single-Precision XX3-form
111100 .... 11100 000...	I	..XX			xvmaxdp	v2.06			777	VSX Vector Maximum Double-Precision XX3-form
111100 .... 11000 000...	I	..XX			xvmaxsp	v2.06			779	VSX Vector Maximum Single-Precision XX3-form
111100 .... 11101 000...	I	..XX			xvmindp	v2.06			781	VSX Vector Minimum Double-Precision XX3-form
111100 .... 11001 000...	I	..XX			xvminsp	v2.06			783	VSX Vector Minimum Single-Precision XX3-form
111100 .... 01110 001...	I	..XX			xvmsubadp	v2.06			716	VSX Vector Multiply-Subtract Type-A Double-Precision XX3-form
111100 .... 01010 001...	I	..XX			xvmsubasp	v2.06			719	VSX Vector Multiply-Subtract Type-A Single-Precision XX3-form
111100 .... 01111 001...	I	..XX			xvmsubmdp	v2.06			716	VSX Vector Multiply-Subtract Type-M Double-Precision XX3-form
111100 .... 01011 001...	I	..XX			xvmsubmsp	v2.06			719	VSX Vector Multiply-Subtract Type-M Single-Precision XX3-form
111100 .... 01110 000...	I	..XX			xvmuldp	v2.06			700	VSX Vector Multiply Double-Precision XX3-form
111100 .... 01010 000...	I	..XX			xvmulsp	v2.06			702	VSX Vector Multiply Single-Precision XX3-form
111100 ..... ///// ..... 11110 1001..	I	..XX			xvnabsdp	v2.06			613	VSX Vector Negative Absolute Double-Precision XX2-form
111100 ..... ///// ..... 11010 1001..	I	..XX			xvnabssp	v2.06			613	VSX Vector Negative Absolute Single-Precision XX2-form
111100 ..... ///// ..... 11111 1001..	I	..XX			xvnegdp	v2.06			614	VSX Vector Negate Double-Precision XX2-form
111100 ..... ///// ..... 11011 1001..	I	..XX			xvnegsp	v2.06			614	VSX Vector Negate Single-Precision XX2-form
111100 .... 11100 001...	I	..XX			xvnmaddadp	v2.06			722	VSX Vector Negative Multiply-Add Type-A Double-Precision XX3-form
111100 .... 11000 001...	I	..XX			xvnmaddasp	v2.06			726	VSX Vector Negative Multiply-Add Type-A Single-Precision XX3-form
111100 .... 11101 001...	I	..XX			xvnmaddmdp	v2.06			722	VSX Vector Negative Multiply-Add Type-M Double-Precision XX3-form
111100 .... 11001 001...	I	..XX			xvnmaddmsp	v2.06			726	VSX Vector Negative Multiply-Add Type-M Single-Precision XX3-form
111100 .... 11110 001...	I	..XX			xvnmsubadp	v2.06			729	VSX Vector Negative Multiply-Subtract Type-A Double-Precision XX3-form
111100 .... 11010 001...	I	..XX			xvnmsubasp	v2.06			732	VSX Vector Negative Multiply-Subtract Type-A Single-Precision XX3-form

Table H.1: Power ISA Instruction Set Sorted by Mnemonic (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
111100 .... 11111 001...	I	..XX			xvnmsubmdp	v2.06			729	VSX Vector Negative Multiply-Subtract Type-M Double-Precision XX3-form
111100 .... 11011 001...	I	..XX			xvnmsubmsp	v2.06			732	VSX Vector Negative Multiply-Subtract Type-M Single-Precision XX3-form
111100 ..... ///// ..... 01100 1001..	I	..XX			xvrdpi	v2.06			808	VSX Vector Round to Double-Precision Integer using round to Nearest Away XX2-form
111100 ..... ///// ..... 01110 1011..	I	..XX			xvrdpic	v2.06			809	VSX Vector Round to Double-Precision Integer Exact using Current rounding mode XX2-form
111100 ..... ///// ..... 01111 1001..	I	..XX			xvrdpim	v2.06			810	VSX Vector Round to Double-Precision Integer using round toward -Infinity XX2-form
111100 ..... ///// ..... 01110 1001..	I	..XX			xvrdpip	v2.06			810	VSX Vector Round to Double-Precision Integer using round toward +Infinity XX2-form
111100 ..... ///// ..... 01101 1001..	I	..XX			xvrdpiz	v2.06			811	VSX Vector Round to Double-Precision Integer using round toward Zero XX2-form
111100 ..... ///// ..... 01101 1010..	I	..XX			xvredp	v2.06			735	VSX Vector Reciprocal Estimate Double-Precision XX2-form
111100 ..... ///// ..... 01001 1010..	I	..XX			xvresp	v2.06			736	VSX Vector Reciprocal Estimate Single-Precision XX2-form
111100 ..... ///// ..... 01000 1001..	I	..XX			xvrspi	v2.06			812	VSX Vector Round to Single-Precision Integer using round to Nearest Away XX2-form
111100 ..... ///// ..... 01010 1011..	I	..XX			xvrspic	v2.06			813	VSX Vector Round to Single-Precision Integer Exact using Current rounding mode XX2-form
111100 ..... ///// ..... 01011 1001..	I	..XX			xvrspim	v2.06			814	VSX Vector Round to Single-Precision Integer using round toward -Infinity XX2-form
111100 ..... ///// ..... 01010 1001..	I	..XX			xvrspip	v2.06			814	VSX Vector Round to Single-Precision Integer using round toward +Infinity XX2-form
111100 ..... ///// ..... 01001 1001..	I	..XX			xvrspiz	v2.06			815	VSX Vector Round to Single-Precision Integer using round toward Zero XX2-form
111100 ..... ///// ..... 01100 1010..	I	..XX			xvrsqrtedp	v2.06			737	VSX Vector Reciprocal Square Root Estimate Double-Precision XX2-form
111100 ..... ///// ..... 01000 1010..	I	..XX			xvrsqrtesp	v2.06			738	VSX Vector Reciprocal Square Root Estimate Single-Precision XX2-form
111100 ..... ///// ..... 01100 1011..	I	..XX			xvsqrtdp	v2.06			704	VSX Vector Square Root Double-Precision XX2-form
111100 ..... ///// ..... 01000 1011..	I	..XX			xvsqrtsp	v2.06			705	VSX Vector Square Root Single-Precision XX2-form
111100 .... 01101 000...	I	..XX			xvsubdp	v2.06			706	VSX Vector Subtract Double-Precision XX3-form
111100 .... 01001 000...	I	..XX			xvsubsp	v2.06			708	VSX Vector Subtract Single-Precision XX3-form
111100 ...// .... 01111 101../	I	..XX			xvtdivdp	v2.06			739	VSX Vector Test for software Divide Double-Precision XX3-form
111100 ...// .... 01011 101../	I	..XX			xvtdivsp	v2.06			740	VSX Vector Test for software Divide Single-Precision XX3-form
111100 ...// 00010 ..... 11101 1011./	I	..XX			xvtlsbb	v3.1			936	VSX Vector Test Least-Significant Bit by Byte XX2-form
111100 ...// ///// 01110 1010./	I	..XX			xvtsqrtdp	v2.06			741	VSX Vector Test for software Square Root Double-Precision XX2-form
111100 ...// ///// 01010 1010./	I	..XX			xvtsqrtsp	v2.06			742	VSX Vector Test for software Square Root Single-Precision XX2-form
111100 .... 1111. 101...	I	..XX			xvtstdcdp	v3.0			872	VSX Vector Test Data Class Double-Precision XX2-form
111100 .... 1101. 101...	I	..XX			xvtstdcsp	v3.0			873	VSX Vector Test Data Class Single-Precision XX2-form
111100 ..... 00000 ..... 11101 1011..	I	..XX			xvxexpdp	v3.0			874	VSX Vector Extract Exponent Double-Precision XX2-form
111100 ..... 01000 ..... 11101 1011..	I	..XX			xvxexpsp	v3.0			874	VSX Vector Extract Exponent Single-Precision XX2-form

Table H.1: Power ISA Instruction Set Sorted by Mnemonic (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
111100 ..... 00001 ..... 11101 1011..	I	..XX			xvxsigdp	v3.0			875	VSX Vector Extract Significand Double-Precision XX2-form
111100 ..... 01001 ..... 11101 1011..	I	..XX			xvxsigsp	v3.0			875	VSX Vector Extract Significand Single-Precision XX2-form
000001 01000 0//// ///// ///// ////// 100001 .... 00....	I	..XX			xxblendvb	v3.1			912	VSX Vector Blend Variable Byte 8RR:XX4-form
000001 01000 0//// ///// ///// ////// 100001 .... 11....	I	..XX			xxblendvd	v3.1			912	VSX Vector Blend Variable Doubleword 8RR:XX4-form
000001 01000 0//// ///// ///// ////// 100001 .... 01....	I	..XX			xxblendvh	v3.1			913	VSX Vector Blend Variable Halfword 8RR:XX4-form
000001 01000 0//// ///// ///// ////// 100001 .... 10....	I	..XX			xxblendvw	v3.1			913	VSX Vector Blend Variable Word 8RR:XX4-form
111100 ..... 10111 ..... 11101 1011..	I	..XX			xxbrd	v3.0			914	VSX Vector Byte-Reverse Doubleword XX2-form
111100 ..... 00111 ..... 11101 1011..	I	..XX			xxbrh	v3.0			915	VSX Vector Byte-Reverse Halfword XX2-form
111100 ..... 11111 ..... 11101 1011..	I	..XX			xxbrq	v3.0			915	VSX Vector Byte-Reverse Quadword XX2-form
111100 ..... 01111 ..... 11101 1011..	I	..XX			xxbrw	v3.0			916	VSX Vector Byte-Reverse Word XX2-form
000001 01000 0//// ///// ///.. ..... 100010 .... 01....	I	..XX			xxeval	v3.1			910	VSX Vector Evaluate 8RR:XX4-form
111100 ..... /.... ..... 01010 0101..	I	..XX			xxextractuw	v3.0			917	VSX Vector Extract Unsigned Word XX2-form
111100 .... 11100 10100.	I	..XX			xxgenpcvbm	v3.1			926	VSX Vector Generate PCV from Byte Mask X-form
111100 .... 11101 10101.	I	..XX			xxgenpcvdm	v3.1			928	VSX Vector Generate PCV from Doubleword Mask X-form
111100 .... 11100 10101.	I	..XX			xxgenpcvhm	v3.1			931	VSX Vector Generate PCV from Halfword Mask X-form
111100 .... 11101 10100.	I	..XX			xxgenpcvwm	v3.1			933	VSX Vector Generate PCV from Word Mask X-form
111100 ..... /.... ..... 01011 0101..	I	..XX			xxinsertw	v3.0			917	VSX Vector Insert Word XX2-form
111100 .... 10000 010...	I	..XX			xxland	v2.06			907	VSX Vector Logical AND XX3-form
111100 .... 10001 010...	I	..XX			xxlandc	v2.06			907	VSX Vector Logical AND with Complement XX3-form
111100 .... 10111 010...	I	..XX			xxleqv	v2.07			907	VSX Vector Logical Equivalence XX3-form
111100 .... 10110 010...	I	..XX			xxlnand	v2.07			907	VSX Vector Logical NAND XX3-form
111100 .... 10100 010...	I	..XX			xxlnor	v2.06			908	VSX Vector Logical NOR XX3-form
111100 .... 10010 010...	I	..XX			xxlor	v2.06			908	VSX Vector Logical OR XX3-form
111100 .... 10101 010...	I	..XX			xxlorc	v2.07			908	VSX Vector Logical OR with Complement XX3-form
111100 .... 10011 010...	I	..XX			xxlxor	v2.06			908	VSX Vector Logical XOR XX3-form
011111 .../// 00000 ///// 00101 10001/	I	....	MMA	MMA	xxmfacc	v3.1			876	VSX Move From Accumulator X-form
111100 .... 00010 010...	I	..XX			xxmrghw	v2.06			918	VSX Vector Merge High Word XX3-form
111100 .... 00110 010...	I	..XX			xxmrglw	v2.06			918	VSX Vector Merge Low Word XX3-form
011111 .../// 00001 ///// 00101 10001/	I	....	MMA	MMA	xxmtacc	v3.1			877	VSX Move To Accumulator X-form
111100 .... 00011 010...	I	..XX			xxperm	v3.0			923	VSX Vector Permute XX3-form
111100 .... 0..01 010...	I	..XX			xxpermdi	v2.06			922	VSX Vector Permute Doubleword Immediate XX3-form
111100 .... 00111 010...	I	..XX			xxpermr	v3.0			923	VSX Vector Permute Right-indexed XX3-form
000001 01000 0//// ///// ///// ///... 100010 .... 00....	I	..XX			xxpermx	v3.1			924	VSX Vector Permute Extended 8RR:XX4-form
111100 .... 11....	I	..XX			xxsel	v2.06			909	VSX Vector Select XX4-form
011111 .../// 00011 ///// 00101 10001/	I	....	MMA	MMA	xxsetaccz	v3.1			877	VSX Set Accumulator to Zero X-form
111100 .... 0..00 010...	I	..XX			xxsldwi	v2.06			925	VSX Vector Shift Left Double by Word Immediate XX3-form
000001 01000 0//// .... 100000 ..... 000..	I	..XX			xxsplti32dx	v3.1			919	VSX Vector Splat Immediate32 Doubleword Indexed 8RR:D-form
111100 ..... 00... ..... 01011 01000.	I	..XX			xxspltib	v3.0			919	VSX Vector Splat Immediate Byte X-form
000001 01000 0//// .... 100000 ..... 0010.	I	..XX			xxspltidp	v3.1			920	VSX Vector Splat Immediate Double-Precision 8RR:D-form

Table H.1: Power ISA Instruction Set Sorted by Mnemonic (continued)

Instruction	Book	Compliancy Subsets	Linux Optional Category	Always Optional Category	Mnemonic	Version	Privilege	Mode Dep	Page	Name
000001 01000 0//// .... 100000 ..... 0011.	I	..XX			xxspltiw	v3.1			920	VSX Vector Splat Immediate Word 8RR:D-form
111100 ..... ///.. ..... 01010 0100..	I	..XX			xxspltw	v2.06			921	VSX Vector Splat Word XX2-form

Last Page – End of Document